8-1-2002

Analysis and hardware implementation of color map inversion algorithms

Michael Martin

Analysis and Hardware Implementation of Color Map Inversion Algorithms

by

Michael W. Martin

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
MASTER OF SCIENCE
Computer Engineering

Approved By:

Principal Advisor _____
Kenneth Hsu, Professor, Computer Engineering

Committee Member _____
Soheil Dianat, Professor, Electrical Engineering

Committee Member _____
Athimoottil Mathew, Professor, Electrical Engineering

Department Head _____
Andreas Savakis, Associate Professor and Department Head, Computer Engineering

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2002

REPRODUCTION PERMISSION STATEMENT

PERMISSION DENIED

Analysis and Hardware Implementation of Color Map Inversion Algorithms

I, Michael W. Martin, hereby **deny permission** to any individual or organization to reproduce this thesis in whole or in part.

Michael W. Martin

8/13/02

Date

Abstract

The purpose of this thesis is to investigate several algorithms that are used to compute the inverse of a forward printer map. The forward printer map models the printer by mapping points in the printer's input color space to points in the printer's output color space. The inverse of this forward map is required to convert input color specifications in a device-independent color space to a color in the printer's device-dependent color space before being presented to the print engine. The accuracy of the inverse printer map directly affects the accuracy of the reproduced colors. Therefore, any measured change in the forward printer map requires re-computation of the inverse map if accurate and consistent color reproduction is to be maintained. An efficient and accurate method of computing the inverse map could be used in an automatic color correction system.

Three algorithms for computing the inverse of the forward printer map are studied in this thesis project. These are the Shepard's, Moving Matrix, and Iteratively Clustered Interpolation (ICI) algorithms. The algorithms are implemented in C and simulated in order to benchmark their relative accuracy, speed, and complexity. The simulations show the ICI algorithm to be the fastest and most accurate at computing the inverse map, and its complexity does not far exceed that of the other algorithms. The ICI algorithm was implemented in VHDL and synthesized to a Synopsys generic library in order to determine the approximate size and speed of an ASIC that could perform the inverse computation. The final implementation resulted in two modules: one that implements the ICI algorithm, and one that implements the trilinear interpolation function that is used by the ICI algorithm. The synthesized ICI module contained 112,683 cells, and the

synthesized trilinear interpolation module contained 190,357 cells. The timing of the modules resulted in a 40 nanosecond clock period, which corresponds to a maximum operating frequency of 25 MHz. These synthesized results show that this algorithm is suitable for an ASIC that could be used in a real-time automatic color correction system.

Table of Contents

List of Figures

List of Tables

1 Introduction

One of the most crucial challenges facing today's printer manufacturers is to design printers that accurately and consistently reproduce the colors in images [1]. This is a difficult task because the transformation from the printer's input color space to its output color space is non-linear and drifts over time. Color reproduction is also affected by the physical properties of the media, including temperature, moisture content, brightness, and weight. These variables are constantly changing and can lead to inaccurate and inconsistent color reproduction. When this occurs, the printer must be re-calibrated. The calibration process is typically manual and time-consuming and reduces the productivity of the printer.

A color correction system is needed to account for these variations in order to accurately and consistently reproduce the colors in printed images. Solutions that can automate the color correction process will lead to very productive printers with little "downtime". There is an abundance of research activity aimed at developing efficient and effective color calibration and control systems [6, 8, 10]. The forward and inverse printer maps are central to many of these systems.

A forward printer map is a practical and accurate model of a printer. This forward map takes the form of a multidimensional look-up table and is constructed by performing input-output color experiments on an actual printer. The inverse of this forward map, called the inverse printer map, is needed to convert an input color specification in a device-independent color space to a color in the printer's device-dependent color space

before being presented to the print engine. If the inverse printer map is accurate, the print engine will produce a color reproduction that closely matches the original device-independent color specification. Any measured change in the forward printer map requires re-computation of the inverse map if color accuracy and consistency is to be maintained. A method for accurately and efficiently computing the inverse of the forward map is sought.

Section 1 of this paper is this brief introduction. In Section 2, several important topics relating to this project will be discussed. Section 3 describes how printers are modeled. Section 4 presents the problem at hand and the theory behind the problem. The inversion algorithms are also discussed in this chapter. Section 5 describes the software implementation and results. Section 6 describes the hardware implementation and results. Finally, Section 7 concludes the paper with a discussion of conclusions, future work, and acknowledgements. References and Appendices can be found at the end of the paper. The CD accompanying this paper includes the paper itself as well as data files, source C and VHDL code, netlists, and area and timing reports of the synthesis results.

2 Background Information

2.1 The Xerographic Printing Process

Imaging systems consist of a network consisting primarily of video display monitors, scanners, and printers. Each of these devices have their own device-specific color space; generally RGB for monitors and scanners and CMY for printers. These devices communicate image data with each other using a psycho-physical based and device-independent color space, such as LAB.

A specification in the device-independent color space is transformed into a device-dependent color space using a multidimensional color correction table or inverse color map. This inverse map is based on the forward transformation characteristics of the printer. Once converted into the device-dependent color space, the color-separated image is converted to halftones and delivered to the print engine that is responsible for the physical production of the print [8]. This process is illustrated in Figure 1.

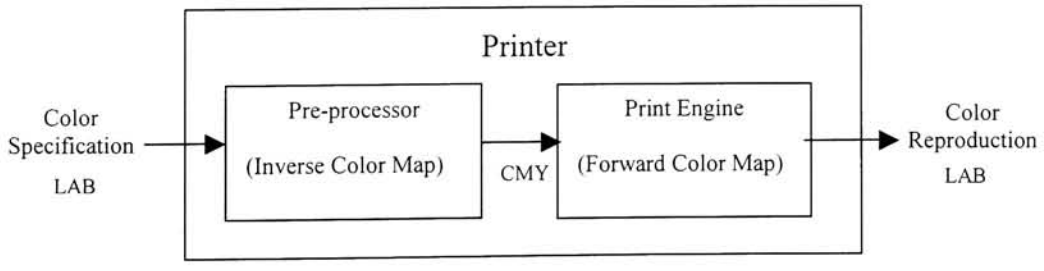


Figure 1: The Xerographic Printing Process

2.2 Color Spaces

Color is specified in a three dimensional space where each point in the space describes a single color. There are several such color spaces, such as RGB, CMY, and LAB. This project focuses on printers; therefore, this research work was done using the CMY and LAB color spaces described in this section. The RGB color space is also described in order to further explain and contrast the additive and subtractive nature of the color systems. Although not used directly in the printing process, RGB is the color system used on the displays that show the images prior to printing and to which the color reproduction is often compared.

2.2.1 RGB Color Space

In the RGB space, a color consists of the primary components red, green, and blue. The RGB system is additive, meaning that varying amounts of the primaries are added together to produce a given color. The additive nature of the RGB system makes it the color system of choice for devices that are self-illuminating light sources, such as displays and scanners. This color space is known as a device-dependent color space because the same RGB color specification will vary perceptually on different devices.

2.2.2 CMY Color Space

Another device-dependent color space is the CMY space. A color in this space consists of the primary components cyan, magenta, and yellow. This system is subtractive,

meaning that varying amounts of the primaries are used together to absorb certain colors; those colors that are reflected, rather than absorbed, combine to form the perceived color. Color prints are illuminated by an external light source, and the perceived colors are due to the combination of reflected wavelengths. Thus, a printer's input color space is typically described in CMY.

Printers typically add a fourth ink – black – which results in four-color CMYK. The black ink is added for several reasons. To produce black using only the CMY primaries, 100 percent of each ink must be used. This is a disadvantage because color inks are more expensive and because using 100 percent of the three primaries results in a very wet print that is susceptible to wrinkles and takes a long time to dry. Furthermore, the combination of full CMY usually results in more of a muddy brown than black color. Also, because the overlay of inks may not be exact, black portions of a print may have edges that do not have full CMY coverage. If these reasons are not enough, including black in the color mix can increase the contrast of colors thereby increasing a printer's color gamut. Replacing some percentage of the three primaries is achieved by gray component replacement and under color removal [7]. A color can be specified using only these primaries; the black component is added later by a separate and independent processing step. In this project, we need only consider the three primaries cyan, magenta, and yellow.

2.2.3 LAB Color Space

The LAB (CIELab) color system was developed by the Commission Internationale de l'Eclairage (CIE) in 1976 [7]. There is a distribution of wavelengths for any given light that is called the spectral power distribution. This system is based on the spectral power distribution of colors. The CIE defined three color-matching functions for humans that are based on the human psycho-physical perception of color. The integral of the spectral power distribution is weighted by the three color-matching functions to provide what is known as the tristimulus values of the color. The LAB system defines a three dimensional color space with tristimulus values L, a, and b. The L component is a measure of lightness of a color. This axis varies from 0 (black) to 100 (reference white). The a component is a chromatic measure of red-green, and the b component is a chromatic measure of blue-yellow. The maximum of a, specified only by +a, is only red with no green, while the minimum of a, specified only by -a, is only green with no red. Similarly, the maximum of b, specified only by +b, is only yellow with no blue, and the minimum of b, specified only by -b, is only blue with no yellow. The axes of this color space are shown in Figure 2.

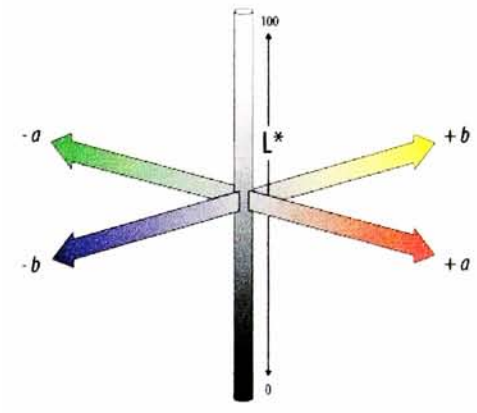


Figure 2: Axes of the CIELAB color space.

The LAB color space is device-independent; color specification using this system are representative only of the color's spectral power distribution and is in no way related to any particular device. Colors with identical spectral power distribution are perceived to be exactly the same color, provided they are seen in the same visual environments, regardless of the manner in which the color is being displayed. This color space is the industry standard for describing colors in a manner that is independent of any device.

2.3 Gamut Mapping

Printers are incapable of reproducing every single color in a given color space. The set or range of colors that a printer can reproduce is known as the printer's color gamut. An input color specification can be located either inside or outside of this gamut. A printer can immediately reproduce a color that is inside the gamut. Colors located outside of the gamut are not reproducible by the printer and must be mapped to a color in the gamut.

This section describes the gamut mapping process that was used in this project [4]. Gamut mapping was necessary after constructing the structured input of the inverse printer map (see Section 4). The process consists of two steps: first determining whether an input color specification (input point) is located inside or outside the gamut, then mapping colors that are outside of the gamut to colors inside the gamut.

The procedure to determine whether an input point is located inside or outside of the printer gamut begins by upsampling the forward printer map P. This 13^3 data set was

upsampled to 64^3 using trilinear interpolation. The upsampling creates a continuous 3D solid with a smooth boundary in the CMY and corresponding LAB color spaces. The boundary points of the CMY structure are assumed to correspond to the boundary points of the LAB structure. This is a practical and acceptable assumption for most xerographic printers.

The input point and the boundary points of the LAB structure are shifted with respect to the centroid of the LAB structure. This centroid is the average of all of the points in the gamut, approximately L=50, a=0, and b=0 for most xerographic printers. These points are then converted to spherical coordinates r, α, and θ. The conversion is given by the following equations:

$$r = \sqrt{(L - L_E)^2 + (a - a_E)^2 + (b - b_E)^2}$$

$$\alpha = \arctan\left(\frac{(b - b_E)}{(a - a_E)}\right)$$

$$\theta = \arctan\left(\frac{(L - L_E)}{\sqrt{(a - a_E)^2 + (b - b_E)^2}}\right)$$

In the above equations, $[L_E\ a_E\ b_E]$ specifies the centroid of the LAB structure. r is the distance from the centroid to the input point. α is the hue angle with range 360°. θ is the angle in a plane of constant α with range 180°.

The set of boundary points is searched for those points that fall into the range $\alpha \pm \Delta\alpha$ and $\theta \pm \Delta\theta$ of the input point. For this project, $\Delta\alpha = \Delta\theta = 5°$. Geometrically, this forms a

cone around the input point. The average distance r is calculated for all boundary points that fall within this cone. If the distance parameter of the input point is greater than this average, then the input point is considered to be outside of the gamut. Conversely, if the input point's distance is less than the average, it is considered to be inside the gamut.

A point outside the gamut should be mapped to a point in the gamut such that the original and mapped points are as perceptually close as possible. Research has shown that the perceptual difference between two colors is lowest when the lightness and hue of the colors are equal [5, 7]. Therefore, the mapping approach used for this project aimed to preserve the lightness and hue of the original point. The boundary of the LAB structure was searched to find the point that best satisfied the mapping criteria described above and given by the follow equations:

$$L \cong L'$$

$$\alpha' = \arctan\left(\frac{b'}{a'}\right) \cong \arctan\left(\frac{b}{a}\right) = \alpha$$

[L a b] specifies the original input point and [L' a' b'] specifies the mapped point. The mapping is illustrated in Figure 3.

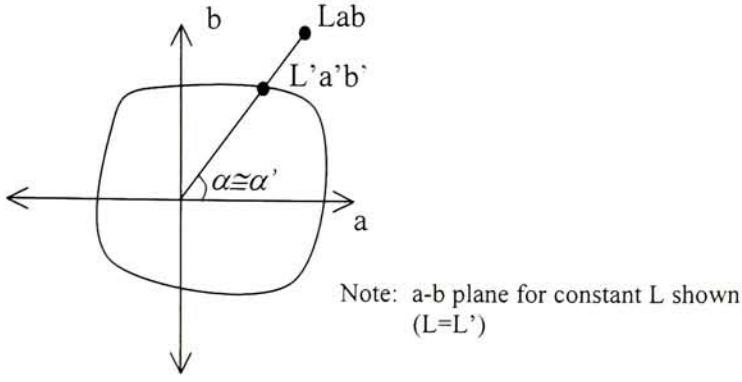


Figure 3: Gamut Mapping for Constant Lightness and Hue

3 Device Modeling

Device color calibration requires a model of the device. These models are generally either theoretical or empirical. The advantage of theoretical models is that color prediction can be performed with relatively few actual measurements. However, these analytical models are usually not very accurate because they do not adequately capture the nonlinearities of real systems due in large part to external variables. Examples of such external variables are temperature, humidity and the weight and type of printable media. These variables have a significant affect on the colors that are reproduced. The Neugebauer equations are a classic example of color prediction using a theoretical model [5].

Polynomial regression is one type of empirical model. This model suffers from accuracy problems; prediction of colors close to the sample points used for the regression provide acceptable accuracy, but for the rest of the points in the gamut there is no guarantee [9].

The most accurate and practical, and therefore the industry standard, empirical model is a look-up table containing input-output pairs that characterize the device. This model is discussed in detail in the following section.

3.1 Forward Printer Map

A color printer can be viewed as a device that maps colors from an input color space to an output color space. The printer can be modeled by a look-up table containing points in the input color space and their corresponding mapping to points in the output color space. The full model can be realized by interpolation for input points not contained in the look-up table. The look-up table is an approximation to the actual printer function; the larger the look-up table, the better the approximation. This look-up table characterizes the printer function and is referred to the forward printer map. The forward printer map is denoted P.

The forward printer map is constructed experimentally by printing colors in the device's input color space on paper and measuring these color patches in the output color space using a spectrophotometer. The disadvantage of this model is that many color experiments must be performed in order to construct a forward map that is an accurate representation of the device. Performing these color experiments is time-consuming,

especially because this process currently non-automated. For example, to cover the entire gamut of a typical printer, a 10x10x10 entry table resulting in 6,000 parameters is needed for acceptable modeling accuracy [6, 8].

The input of the forward printer map can be structured by sampling the printer's input space in equal steps along each axis of the source domain. These sampled points become the "node", "grid", or "lattice" points of the forward map. For n levels of division, this gives $(n-1)^3$ cubes and n^3 lattice points. The n^3 lattice points of the source space are printed, and these color patches are measured to determine the output color space specification. The corresponding values from the source and destination spaces populate the look-up table. Note that the output of P will be unstructured (or non-uniform) because the transformation from input color space to output color space is typically highly nonlinear.

3.2 Inverse Printer Map

The inverse printer map, denoted P^{-1}, is constructed from the forward printer map P. The inverse map associates points in the output color space with points in the device's input color space. Simply swapping the data from P is not desirable for two reasons. First, the inverse look-up table will not be well defined for colors near the gamut boundary because it is possible that more than one input point be mapped to the same output point. Second, it results in a look-up table with unstructured input. Interpolation with data that has unstructured input is complex and time-consuming. On the other hand, if the input is structured, very fast and efficient geometric interpolation methods can be utilized.

Geometrical linear interpolation is preferred over other non-linear interpolation techniques because it is more efficient and accurate. There are several geometrical interpolations, including trilinear, prism, pyramid, and tetrahedral [5]. Each of these vary in terms of accuracy, extraction (or search) complexity, and interpolation (or computation) complexity. In this thesis project, trilinear interpolation was utilized for look-up table interpolation (see Section 4.1.4).

Instead of simply swapping the data from P, algorithms are required that interpolate irregularly sampled multi-dimensional data from P. These algorithms must be capable of efficiently computing the inverse map so that it has structured input points and so that it is as best an approximation to the true inverse as possible. Several of these algorithms are discussed in the next section.

4 Theory

Consider the forward printer map P that maps a point y in the printer's input color space to a point z in the output color space. Also consider the inverse printer map P^{-1} that maps a point x in the target color space to a point $\tilde{y}$ in the printer's input color space. Given a point x in the target color space, the inverse color map is used to find a point y in the printer's input color space that the printer will map to a point z in the output color space. This system is depicted in Figure 4.

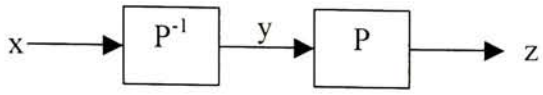


Figure 4: System of P^{-1} and P

The desired result is for the printer to print a color z that, when measured, will be as close as possible to the requested color x. The difference between the target and output colors can be quantified by computing the Euclidean distance between them. This difference is known as $\Delta E(P(y),x)$ or simply ΔE and is given by Equation 1.

$$\Delta E = \|z - x\| = \|P(y) - x\| \tag{1}$$

Note that

$$z = P(y)$$

In general, and specifically for this project, the target and output color space is the device-independent LAB color space. For the target color $(LAB)_{in}$ and the output color $(LAB)_{out}$, Equation 1 becomes:

$$\Delta E = \sqrt{(L_{out} - L_{in})^2 + (A_{out} - A_{in})^2 + (B_{out} - B_{in})^2}$$

The smaller the value of ΔE for a given input, the better the inverse map. Humans do not perceive color differences less than 1.0. ΔE = 1.0 is know as the "just noticeable color difference." This leads to the following optimization problem:

Given a target color x and the forward printer map P, find the printer input y that solves

$$\min_{y} \Delta E(P(y), x) \tag{2}$$

Solving this optimization problem results in the inverse printer map P^{-1}. The structured inverse printer look-up table is constructed according to the methodology outlined in [1], re-stated below:

1. Obtain the estimated forward printer map, P, of a given color printer. This is achieved by equally sampling the printer input space Y, and then the printed color z_j for each grid node y_j is obtained from experiments on the actual printer. In addition, some interpolation technique is used to estimate the outputs corresponding to input points that are not grid nodes.

2. Grid the target color space X to obtain an ordered collection of vectors x_i, i = 1, 2, ..., h (each x_i corresponds to a grid node). This will result in grid nodes that are outside of the printer gamut; these points must be first mapped onto the gamut using the approach described in Section 2.3.

3. Obtain the y_i that solves Equation 2 for x_i and P, for i = 1, 2, ..., h.

4. Select a mutidimensional interpolation technique to obtain the inverse map outputs y corresponding to input points x that are not exactly grid nodes.

Several algorithms have been proposed for solving the optimization problem given by Equation 2. These algorithms can vary widely in terms of accuracy, numerical stability, speed, and complexity. Three of these algorithms are Shepard's Interpolation [3], Moving Matrix [2], and Iteratively Clustered Interpolation [1]. These algorithms were the subject of study in this thesis. The details of each algorithm are discussed in the following sections. Trilinear interpolation is used frequently by the ICI algorithm to interpolate through P and P^{-1}; it is discussed at the end of this section.

4.1 Algorithms

The algorithms will be presented in an abstract sense, where x represents colors in the target color space, y represents colors in the printer's input color space, and z represents colors in the printer's output color space.

For the algorithms discussed in this section, recall that he forward printer map P contains N number of entries and defines the mapping function:

$$z_j = P(y_j) \quad \text{for } 0 < j < N$$

For the forward printer map used in this project, the printer's input color space was CMY, and its output color space was LAB. The algorithms can be applied to this specific application by remembering that x, y, and z are equivalent to the color space vectors:

$$x = [L\ A\ B]$$

$$y = [C\ M\ Y]$$

$$z = [L\ A\ B]$$

4.1.1 Shepard's Algorithm

Shepard's algorithm is based on the work of Donald Shepard and is a well-established method for interpolating scattered data [3]. It is essentially an application of weighted averaging. The value for a given input point is calculated by a weighted average of all other data points, where the weighting is a function of the distance from the given point to the other data points.

Given a color x in the target color space, Shepard's algorithm computes the inverse $\tilde{y}$ by the following equation:

$$\tilde{y} = \begin{cases} \dfrac{\sum_{j=1}^{N} y_j (d_j)^{-\mu}}{\sum_{j=1}^{N} (d_j)^{-\mu}} & \text{if } d_j \neq 0 \text{ for all } z_j \\ y_j & \text{if } d_j = 0 \text{ for some } z_j \end{cases}$$

where d_j is the Lp norm between the input point x and z_j:

$$d_j = \left\| z_j - x \right\|_p$$

There are two variable parameters in this algorithm: p and μ. The p parameter specifies how the distances between points are calculated. The μ parameter affects the locality of the weighting function; large values of μ result in more local behavior, which means that only those points closest to the input point will be significant.

4.1.2 Moving Matrix Algorithm

The Moving Matrix algorithm computes the printer inverse using linear weighted least-squares regression [2]. The inverse is given by the following equation, where A is the transformation matrix.

$$\tilde{y} = xA^T \tag{3}$$

The transformation matrix A is found by minimizing the weighted squared error given by the following equation:

$$\mathrm{E} = \sum_{j=1}^{N} W_j \left\| y_j - Az_j \right\|^2 \tag{4}$$

Differentiating Equation 4 with respect to A, setting it equal to zero, and solving for A yields the closed form solution:

$$A = S_2 S_1^{-1}$$

where

$$S_1 = \sum_{j=1}^{N} W_j z_j^T z_j \quad \text{and}$$

$$S_2 = \sum_{j=1}^{N} W_j y_j^T z_j$$

The output $\tilde{y}$ can then be calculated according to Equation 3.

The weighting is a function of the distances from the input point x to all of the other points z_j contained in the forward printer map:

$$W_j = \frac{1}{{d_j}^{\mu} + \varepsilon}$$

Here d_j is the Euclidean distance from the input point x to the point z_j.

The variable parameters of this algorithm are μ and ε. These parameters affect the locality of the regression. Large values of μ and small values of ε give W_j more local behavior, which means that only those points closest to the input point will be significant.

4.1.3 Iteratively Clustered Interpolation Algorithm

The Iteratively Clustered Interpolation (ICI) algorithm is a gradient-based optimization method that uses an iterative technique to generate initial points [1].

An unconstrained gradient-based optimization algorithm to solve Equation 2 would be given by [11]:

$$y(k+1) = y(k) - \beta\left(\frac{\partial \Delta E(y,x)^2}{\partial y}\right)_{y=y(k)} \qquad k \geq 0, \text{ for a given } y(0) \qquad (5)$$

Note that $\Delta E(y,x)$ is Equation 1, restated below:

$$\Delta E = \|z - x\| = \|P(y) - x\|$$

Thus,

$$\Delta E(y,x)^2 = \|P(y)-x\|^2 = (P(y)-x)'(P(y)-x)$$

The gradient of $\Delta E(y,x)^2$ with respect to y for a fixed x is then:

$$\frac{\partial \Delta E(y,x)^2}{\partial y} = 2J(y)'(P(y)-x) \tag{6}$$

Substituting Equation 6 into Equation 5 gives the following update equation:

$$y(k+1) = y(k) - \mu J_k'(P(y(k))-x) \tag{7}$$

where $\mu=2\beta$. $P(y(k))-x$ is a component vector of the differences between the target color x and the output color z produced by interpolating $y(k)$ through the forward printer map P:

$$P(y(k))-x = \begin{bmatrix} z_{k,0}-x_0 \\ z_{k,1}-x_1 \\ z_{k,2}-x_2 \end{bmatrix}$$

J_k is the Jacobian of $(P(y(k))-x)$ evaluated at $y = y(k)$, and is given by:

$$J_k' = \begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_2}{\partial y_1} & \frac{\partial z_3}{\partial y_1} \\ \frac{\partial z_1}{\partial y_2} & \frac{\partial z_2}{\partial y_2} & \frac{\partial z_3}{\partial y_2} \\ \frac{\partial z_1}{\partial y_3} & \frac{\partial z_2}{\partial y_3} & \frac{\partial z_3}{\partial y_3} \end{bmatrix}$$

This matrix is also known as the gradient matrix. It is calculated using numerical differentiation of the forward printer map P. ∂z is an average of the forward and backward finite differences that result from separately varying each component of y and interpolating through P.

Gradient-based optimization methods suffer from drawback that a solution may be optimal only in a local sense, rather than being the global optimum. In order to avoid this situation, the initial estimation of the inverse must be close to the actual globally optimal solution. A novel and efficient procedure to determine a "good" initial estimate is presented in [1] and is re-stated as follows:

1. Search the z_j points of P for the one that is closest to the given target point x. Call this point z_{aux}.

2. Find the corresponding point y_{aux}. This is the point such that $z_{aux} = P(y_{aux})$. Y_{aux} is a node point in the grid of the Y space, which is a point in the input space of the forward printer map P. This y_{aux} is a course estimate for the inverse of x.

3. Select M points in the neighborhood of y_{aux}; generate a cluster of M points (y_{aux1}, y_{aux2}, ..., y_{auxM}) by moving along axes around y_{aux}. Map these M points to obtain $z_{auxj} = P(y_{auxj})$, j = 1, 2, ..., M.

4. Find the closest of the z_{auxj} points to the input x and call this point z_0. Take the initial estimate $y(0)=y_0$ to be the point such that $z_0 = P(y_0)$.

After using this procedure to find the initial estimate *P(0)*, the update equation given by Equation 7 can be used. Define parameters k_{max} and ε that are the stop criteria of the iteration of the update equation.

ε specifies the error threshold at which the update iterations can stop, such that

$$\|P(y(k)) - x\| \leq \varepsilon \tag{8}$$

k_{max} is the maximum number of iterations that algorithm will perform when computing the inverse for a given point. It is useful to define this parameter as a stop criteria because not every input point will result in an error that is below the threshold ε. When either Equation 8 is satisfied or $k \geq k_{max}$, the algorithm stops and *y(k)* is taken as the solution.

The parameter δ for this algorithm specifies the perturbation used in the numerical differentiation method used to compute the gradient matrix.

The parameter μ should be selected in order to achieve fast convergence and to meet accuracy requirements. This parameter should be bounded by the following equation, as discussed in [1]:

$$0 < \mu < \frac{2}{trace(J_0 J_0')}$$

4.1.4 Trilinear Interpolation

Trilinear interpolation of a look-up table is a three-dimensional geometric method for computing the output values of input points that are not contained in the look-up tables [5]. For the sake of this discussion, the look-up table is either the forward or inverse printer map, and this interpolation technique is used with these maps for input points that are not grid points.

Trilinear interpolation consists of two steps: the extraction step and the interpolation step. In the extraction step, the sub-cube in the source color space that contains the input point is determined by a series of comparisons. The eight vertices of this cube are the lattice points in the source space, as shown in Figure 5.

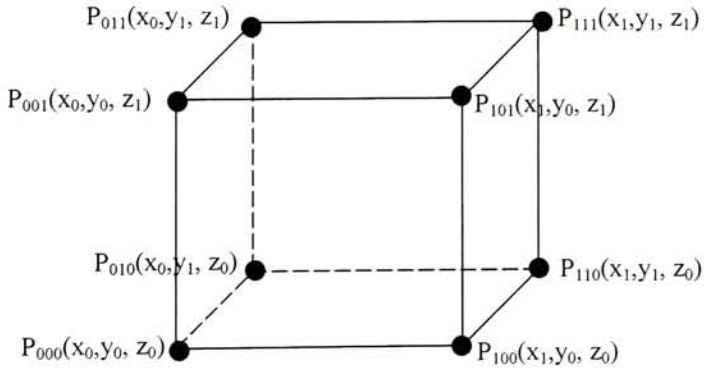


Figure 5: Sub-cube with Lattice Points Determined During Extraction

The interpolation step consists of the repeated use of one-dimensional linear interpolation. The following discussion is a brief review of 1D linear interpolation.

Referring to Figure 6, a point p on the curve between lattice points p_0 and p_1 is to be interpolated.

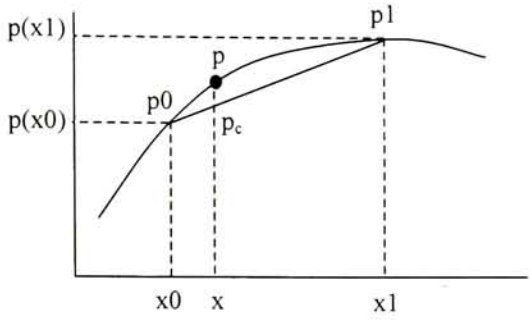


Figure 6: 1D Linear Interpolation

The interpolated value, $p_c(x)$, is linearly proportional to the ratio $(x\text{-}x_0)/(x_1\text{-}x_0)$. Therefore,

$$p_c(x) = p(x_0) + \frac{(x - x_0)}{(x_1 - x_0)}[p(x_1) - p(x_0)]$$

Three-dimensional trilinear interpolation consists of seven linear interpolations on the sub-cube depicted in Figure 5. This results in the following equations [5]:

$$\Delta x = x - x_0$$
$$\Delta y = y - y_0$$
$$\Delta z = z - z_0$$

$$c_0 = p_{000}$$
$$c_1 = (p_{100} - p_{000}) / (x_1 - x_0)$$
$$c_2 = (p_{010} - p_{000}) / (y_1 - y_0)$$
$$c_3 = (p_{001} - p_{000}) / (z_1 - z_0)$$
$$c_4 = (p_{110} - p_{010} - p_{100} + p_{000}) / [(x_1 - x_0)(y_1 - y_0)]$$
$$c_5 = (p_{101} - p_{001} - p_{100} + p_{000}) / [(x_1 - x_0)(z_1 - z_0)]$$
$$c_6 = (p_{011} - p_{001} - p_{010} + p_{000}) / [(y_1 - y_0)(z_1 - z_0)]$$
$$c_7 = (p_{111} - p_{011} - p_{101} - p_{110} + p_{100} + p_{001} + p_{010} - p_{000}) / [(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)]$$

$$p(x,y,z) = c_0 + c_1\Delta x + c_2\Delta y + c_3\Delta z + c_4\Delta x\Delta y + c_5\Delta x\Delta z + c_6\Delta y\Delta z + c_7\Delta x\Delta y\Delta z$$

5 Software Implementation

Software programs implementing the Shepards, Moving Matrix, and ICI algorithms were written in the C programming language. The programs read data files that contained the forward printer map and the gamut-mapped input points of the inverse map. The programs executed the algorithms on this data to compute the inverse for the given input points; this output data was written to separate data files for later analysis. The forward printer map was represented by two data files: one had 13^3 CMY entries, and the other had the corresponding 13^3 LAB entries. This experimental printer data was obtained from Dr. L. K. Mestha from Xerox.

5.1 Algorithm Metrics

These software implementations and simulations provided relative accuracy, execution time, and computational complexity information for each algorithm.

5.1.1 Determining Algorithm Complexity

The complexity of the algorithms was measured objectively by including code to count the number of multiplication/division and square root operations performed to compute the inverse. These operations are costly in hardware because the modules to perform them are both large and slow as compared to other operations. The algorithms were also studied from a subjective standpoint to ascertain a general idea of the number of required

arithmetic modules, the amount of parallelism that might be exploited, and the complexity of the required control logic. A combination of the objective and subjective complexity was used to rate each algorithm's overall complexity relative to the others.

5.1.2 Determining Algorithm Execution Speed

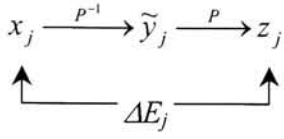
The csh command interpreter running on the Unix platform has a built-in utility that can be used to determine the amount of time a program is executing on the system. The execution times of the programs were measured using this utility, which is an executable named *time*. In order to form a valid comparison, the programs were executed on the same machine under the same environment. Program development and execution was on Rochester Institute of Technology's Grace computing system. Grace is an Alpha 4100 5/533 with 3 EV5.6 CPUs. It has 1.5GB of memory and is running Tru64 UNIX, version 4.0F.

5.1.3 Determining Algorithm Accuracy

The programs computed the inverse for the 13^3 gamut-mapped LAB points x using the forward printer map data. The completion of the programs resulted in a data file containing 13^3 CMY entries that defined the inverse $\tilde{y}$:

$$\tilde{y} = P^{-1}(x)$$

Each point $\tilde{y}_j$ was interpolated through the forward printer map P to find the resulting z_j. The differences ΔE_j between z_j and the corresponding x_j such that $\tilde{y}_j = P^{-1}(x_j)$ were computed. This can be shown graphically as:



The mean error and standard deviation, the minimum error, and the maximum error were calculated from the ΔE_j – these statistics represented the accuracy of the algorithms. It should be noted that these error statistics were computed using the node points of the inverse map as inputs. The accuracy metrics of each algorithm were used to compare the algorithms in terms of accuracy. Separate programs were written to perform the interpolation through P and to compute the accuracy metrics; these are also included on the CD accompanying this paper.

5.2 Optimal Parameters

Each algorithm was executed several times while varying the algorithm parameters. Accuracy metrics for each trial were measured to gain insight into how the parameters affect the accuracy. These trials also led to the values of the parameters that result in the best accuracy. Tables showing the accuracy data for different parameter values for each algorithm are shown in the Appendix (see Section 9.1). The results of these trials are briefly described next.

5.2.1 Shepard's Algorithm

For Shepard's algorithm, there are the two variable parameters p and μ. The p parameter specifies how the distances between points are calculated. The trials showed that the value of p does not have much affect on the accuracy. Therefore, $p = 2$ was taken because this is the most convenient as it results in a distance measurement that is simply the Euclidean distance. The μ parameter affects the locality of the weighting function. The simulation results show that $\mu = 5$ provides the best accuracy.

5.2.2 Moving Matrix Algorithm

The variable parameters in the Moving Matrix algorithm are μ and ε. Similar to the same parameter in Shepard's algorithm, the μ parameter affects the locality of the regression. Simulations showed that $\mu = 5$ resulted in the best accuracy. The ε parameter does not have a significant affect on the accuracy and is taken as a small value to avoid ill-conditioning of the S_1 matrix. In these simulations, ε was taken as 10^{-4}.

5.2.3 ICI Algorithm

The ICI algorithm has parameters ε, k_{max}, δ, and μ. ε and k_{max} are the stop criteria; ε specifies the error threshold at which the update iterations can stop, and k_{max} is the maximum number of iterations that algorithm will perform when computing the inverse for a given point. The parameter δ for this algorithm specifies the perturbation used in the numerical differentiation method used to compute the gradient matrix. The parameter

μ should be selected in order to achieve fast convergence and to meet accuracy requirements.

The software simulations showed that the mean error for this algorithm can be reduced by decreased the error threshold ε. In order to achieve fast convergence, ε was taken as 0.5. This is below the "just noticeable difference" of $\Delta E = 1.0$. Other simulations have shown that even better accuracy can be obtained by further reducing ε [4]. The software simulations also showed that the best choice of k_{max} is 50. Increasing this parameter beyond 50 does not result in substantial gains in accuracy; if the algorithm is going to converge for a given input point, it will do so within 50 iterations. The perturbation δ was determined to not have any measurable affect on the accuracy of the algorithm; δ was taken to be 10, which is approximately a 4% variance of the printer's input color components ($0 < C, M, Y < 255$ and 10/255=0.04) and is more or less arbitrary. Finally, the simulations showed $\mu = 4$ to be a good choice in terms of accuracy and convergence.

Table 1 shows the optimal parameters found for each algorithm.

Table 1: Optimal Algorithm Parameters

Shepard's	$P = 2$ $\mu = 5$
Moving Matrix	$\mu = 5$ $\varepsilon = 10^{-4}$
ICI	$\varepsilon = 0.5$ $k_{max} = 50$ $\delta = 10$ $\mu = 4$

5.3 Simulation Results

The accuracy, speed, and complexity of each algorithm is summarized in Tables 2, 3, and 4, respectively. The metrics are obtained from the simulations using optimal algorithm parameters and operating on the same data and on the same machine in the same environment. All measurements are made for the algorithms computing the 13^3 inverse map from the 13^3 forward map.

Table 2: Algorithm Accuracy

	Mean ΔE	Std Dev	Mean + 2*StdDev	Min	Max
Shepard's	1.64	0.54	2.73	0.15	3.81
Moving Matrix	1.28	2.29	5.86	0.02	54.88
ICI	0.39	0.12	0.63	0.05	1.68

Table 3: Algorithm Execution Time

	Execution Time (sec)
Shepard's	12
Moving Matrix	17
ICI	7

Table 4: Algorithm Complexity

Shepard's	LOW
Moving Matrix	HIGH
ICI	MEDIUM

ICI is clearly the winner in terms of accuracy and execution speed. A brief discussion of each algorithm's complexity follows.

Moving Matrix has significantly more multiplications than Shepard's and approximately five times more multiplications than ICI; this would lead to a comparatively large and/or

slow design. Moving Matrix also would require complex control logic to perform matrix operations such as matrix inversion.

Shepard's algorithm would be quite easy to implement because it requires a small number of resources and is data flow oriented thus requiring very little control logic. However, there are a large number of multiplications (three times that of ICI), which will result in a slow design. Even if more resources were used to perform some of the processing in parallel (which would increase the design area), it is still unlikely it would beat out ICI in terms of speed. Although easy to implement, Shepard's is inaccurate and would most likely be slower in hardware than ICI.

ICI has the smallest number of multiplications, which could result in a less complex circuit. However, ICI has two significant drawbacks. First, this algorithm requires trilinear interpolation, which itself is a non-trivial hardware implementation. Second, ICI will require control logic for computing the gradient matrix and for computing the initial estimate by the clustering technique.

Of the three algorithms, ICI is the best in terms of speed, accuracy, and hardware complexity. This algorithm was chosen for hardware implementation in VHDL. The results of this phase of the project are presented in the next section.

6 ICI Hardware Implementation

NOTE: All source VHDL files are located on the accompanying CD under the "Hardware/VHDL Source Code" folder. The area and timing reports for each sub-module described in this section can also be found on the accompanying CD under the "Hardware/Reports" folder. VHDL source code is also shown in the Appendix (see Section 9.2).

6.1 Design Methodology

The algorithm was implemented using VHDL. The design was simulated with Mentor Graphic's ModelSim; the simulation results were compared with results obtained from software models operating on the same input data. Correct functionality of the design was guaranteed since the hardware simulation results matched those of the software models.

The hardware modules were synthesized using Synopsys's Design Compiler and was targeted to a generic library provided with the Synopsys's tools. The resulting synthesis results provide a net-list that can be used for ASIC fabrication.

Several important design decisions were made about the system partitioning and architecture, data storage, and data representation. These decisions will be described in the following sections.

6.1.1 System Partitioning and Architecture

The ICI algorithm makes extensive use of trilinear interpolation. Trilinear interpolation is a geometric method to calculate the output of a 3D function at a given input point, where the 3D function is defined by a finite set of input and output pairs. A detailed discussion of trilinear interpolation can be found in Section 4.1.4. An entirely separate module was designed to perform trilinear interpolation; this module is used by the ICI design and can also be used in other applications.

Both the trilinear interpolation module and ICI module were designed using datapath blocks and control blocks. This approach allows for highly controlled use of the arithmetic units that make up the datapaths. The advantage of having such control over these units is that operations can be pipelined and the units can be shared where appropriate.

6.1.2 Data Storage

There are six values for each entry in the forward printer map *P*; one for each of the input space components and one for each of the output space components. For example, if the *P* defines a mapping from CMY to LAB, then there is a value for C, M, and Y for the input color space, and a value for L, A, and B for the output color space. An N^3 look-up table requires $6N^3$ bytes of storage. For N=13, this results in 13,182 bytes of required storage.

Synthesis tools are still incredibly inefficient at synthesizing memory; therefore, this design assumes that an external memory module exists that contains *P*. The memory module in Figure 7 is assumed for design purposes. The interpolation and ICI modules are designed to interface with this memory module.

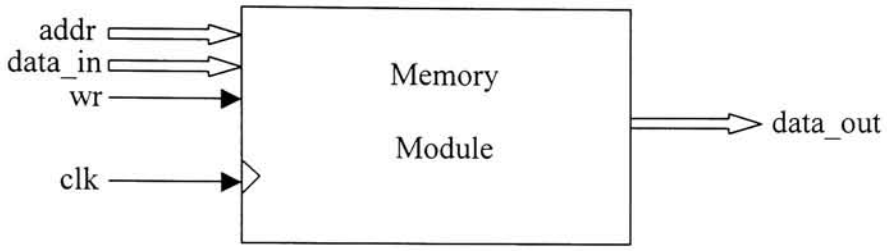


Figure 7: Memory Module Diagram

6.1.2.1 Operation

The memory module is synchronous and clocked by the **clk** input signal; all reads and writes occur on the rising edge of **clk** and complete in one clock cycle. The operation of the module is described below.

(a) Read: Reads occur when **wr** is low; **data_out** becomes the value stored at the location specified by **addr**.

(b) Write: Writes occur when **wr** is high; the location specified by **addr** is stored with the value specified by **data_in**, and **data_out** is the same value as **data_in**.

(c) Memory location: **addr** specifies the memory location to read from during reads and the location to write to during writes. Its value can range from zero to $6N^3$-1.

The interpolation and ICI modules that interface to this memory never write to it; the **wr** and **data_in** signals are included for completeness.

6.1.2.2 Memory Data Layout

It is assumed that the memory module contains the foward printer look-up table prior to using the functionality of the interpolation and ICI modules. Furthermore, it is assumed that the values are arranged in the memory module in the following fashion:

Memory Location	Contents
0	C_0
1	M_0
2	Y_0
3	L_0
4	A_0
5	B_0
6	C_1
7	M_1
8	Y_1
9	L_1
10	A_1
11	B_1
...	...

6.1.3 Data Representation

The project specification required that two decimal places of accuracy be retained for the LAB values. This meant that the design would have to implement floating point operations. Floating point hardware is more complex and slower compared to fixed point hardware. Additionally, the design resources did not have floating-point libraries that contain pre-compiled and optimized floating-point modules. These libraries would either have to be purchased, or a significant amount of time and effort would be required to design custom floating-point arithmetic modules.

An alternative is to use fixed point arithmetic and scale all of the data to integers. The approach allows for the use of pre-compiled and optimized fixed point integer modules, which simplifies the design and reduces design time.

To preserve two decimal places, the data had to be scaled by a factor of 100. For example, 123.45 would be scaled as 123.45*100 = 12345. Integer addition and subtraction would occur normally. However, multiplications would require the result to be scaled down. This can be shown by the following equations:

Original Operation: $A * B = AB$

With Scaled Data: $(100A) * (100B) = 10000AB$

However, we only want 100AB, which means this result must be scaled down by a factor of 100. This requires the result to be divided by 100. This is a disadvantage of using

scaled fixed point arithmetic; multiplication operations must be followed by division. This disadvantage is a fair trade-off compared to the complexity involved in implementing a design to support floating-point numbers.

It was necessary to determine if using this integer-based approach would compromise the accuracy of the ICI algorithm. The ICI C language implementation files were modified to use only integer data in order to investigate the issue. The integer based ICI C code is located on the accompanying CD under the "Software/Integer-based ICI" folder. Simulation results showed that the accuracy of the integer-based implementation was equal to its floating-point-based counterpart.

6.2 Synthesis Methodology

The control blocks of the trilinear interpolation and ICI modules are the only synchronous (clocked) blocks in each of the designs. These blocks were synthesized for speed, and the maximum path delay through them defined the clock and maximum operating frequency for the entire design. All other blocks are asynchronous, and their maximum path delays are either shorter or longer than the defined clocked. For those blocks that have a shorter maximum path delay, there is no concern because the circuit will provide valid results within the clock period. For those blocks that have a longer maximum path delay, the control blocks must either wait the appropriate amount of clock cycles for the results to become valid or continue on with processing. In the latter case, computation in the block would continue as the control block would proceed with other computations using other datapaths. Results would become valid at some later point in

the logic flow. This concept of parallel computation motivated the design of several separate datapaths that perform different functions. Wherever this type of parallelism could not be exploited, the control block simply waits the appropriate amount of clock cycles for valid data.

There were no design specifications, such as area or speed, provided by the project requirements for the hardware implementation. However, the design has application in real-time and near real-time applications. It was also designed for ASIC implementation in mind. For these reasons, the design blocks were synthesized for the fastest possible performance. This is a reasonable synthesis constraint since the goal is real-time computation, and the ASIC platform allows for larger and more complex designs as compared to programmable logic, such as field programmable gate arrays (FPGA).

The asynchronous blocks were constrained by specifying input and output delays equal to the defined clock. These constraints forced the synthesis tool to generate the fastest possible design, but one that would never meet the timing constraints because of the complexity of the operations being performed. Several of the synthesized blocks' timing reports clearly show that they do not meet timing constraints. This is acceptable because in this design methodology it is understood that these blocks require more than one clock cycle to complete.

6.3 Simulation & Testing

The VHDL code was simulated using Mentor Graphic's ModelSim tool. Testbenches were written in order to test and verify correct functionality of the hardware design. These testbenches read input data from files, fed it to the module under test, and wrote the output data into another data file. Other inputs were also controlled by the testbenches; these inputs, such as algorithm parameters, were the same as those used in the C language software simulations. The output data files written by the testbenches were compared to the output data files written by the C software models. Correct functionality of the VHDL design was guaranteed because the data contained in the output files matched. The VHDL source code for the testbenches can be found in the Appendix (see Section 9.2).

6.4 Hardware Modules

6.4.1 Trilinear Interpolation Module (tri_top.vhd)

Figure 8 shows the I/O signals of the trilinear interpolation module, and Figure 9 shows the block diagram of the module. Input and output signals are described in Table 5.

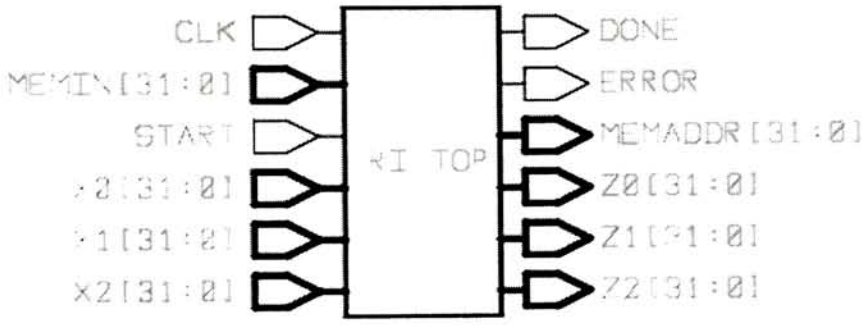


Figure 8: Top Level Diagram of Trilinear Interpolation Module

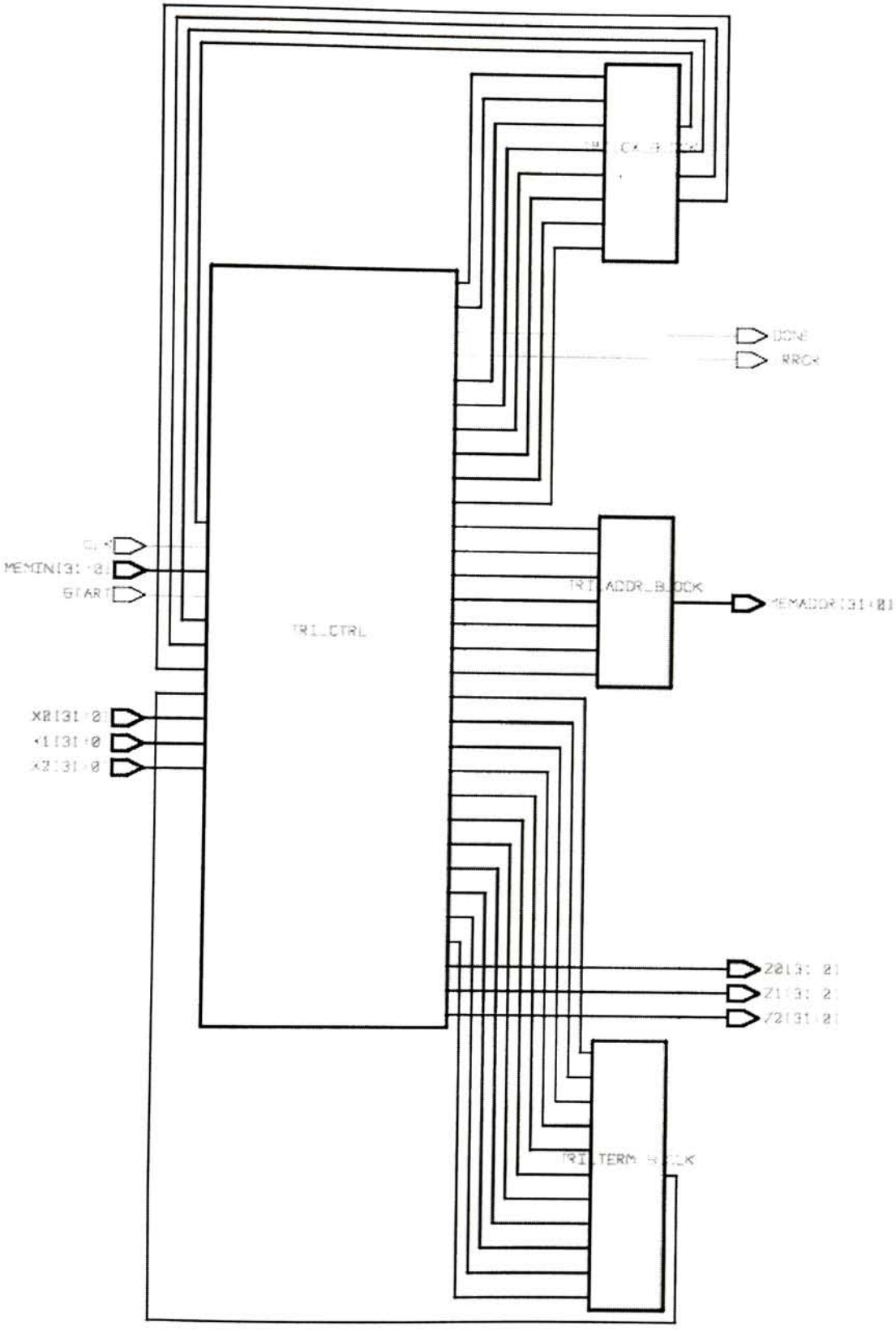


Figure 9: Submodule Diagram of Trilinear Interpolation Module

Table 5: Trilinear Interpolation Module I/O Signals

Signal Name	I/O	Type	Description
Clk	Input	std_logic	System clock.
Memin	Input	std_logic_vector[31:0]	Input data from external memory.
Start	Input	std_logic	Interpolation operation begins on the rising edge of this signal.
X0	Input	std_logic_vector[31:0]	Specifies the first component of the input point.
X1	Input	std_logic_vector[31:0]	Specifics the second component of the input point.
X2	Input	std_logic_vector[31:0]	Specifies the third component of the input point.
Done	Output	std_logic	Signals the completion of the interpolation operation and valid output.
Error	Output	std_logic	Signals an error occurred during the interpolation operation. Note: An error can occurs if the input point specified by X0, X1, and X2 is located outside of the input space defined in the LUT.
Memaddr	Output	std_logic_vector[31:0]	Specifies the location of the desired data in memory.
Z0	Output	std_logic_vector[31:0]	First component of the output value; valid when done=1 and error=0.
Z1	Output	std_logic_vector[31:0]	Second component of the output value; valid when done=1 and error=0.
Z2	Output	std_logic_vector[31:0]	Third component of the output value; valid when done=1 and error=0.

6.4.1.1 Operation

The system clock must have a period of 40 nanoseconds or greater; this corresponds to a maximum operating frequency of 25 MHz. Simulation results show that this design can perform trilinear interpolation for an input point in 2.5 μs when operating at 25 MHz.

The rising edge of start begins interpolation at the input point specified by signals **X0**, **X1** and **X2**. **Done** is asserted when interpolation is complete. If **error** is not raised, valid output occurs on signals **Z0**, **Z1** and **Z2**. **Memin** and **memaddr** are interface signals to the external memory module. **Memin** is data coming from the memory; **memaddr** is the location in memory of desired data and is controlled by the interpolation module's control block.

6.4.1.2 Sub-blocks

6.4.1.2.1 TRI_CTRL_BLOCK (tri_ctrl.vhd)

This block contains the control logic needed to carry out the interpolation. Its input and output signals interface to the other sub-blocks and to the external memory module to control data flow. It is also responsible for beginning the interpolation when start is raised and for asserting done and error when the interpolation is complete.

The top level diagram is shown in Figure 10. A lower level block diagram is not available because it contains too many low-level components (primarily registers) and interconnects to be displayed on paper.

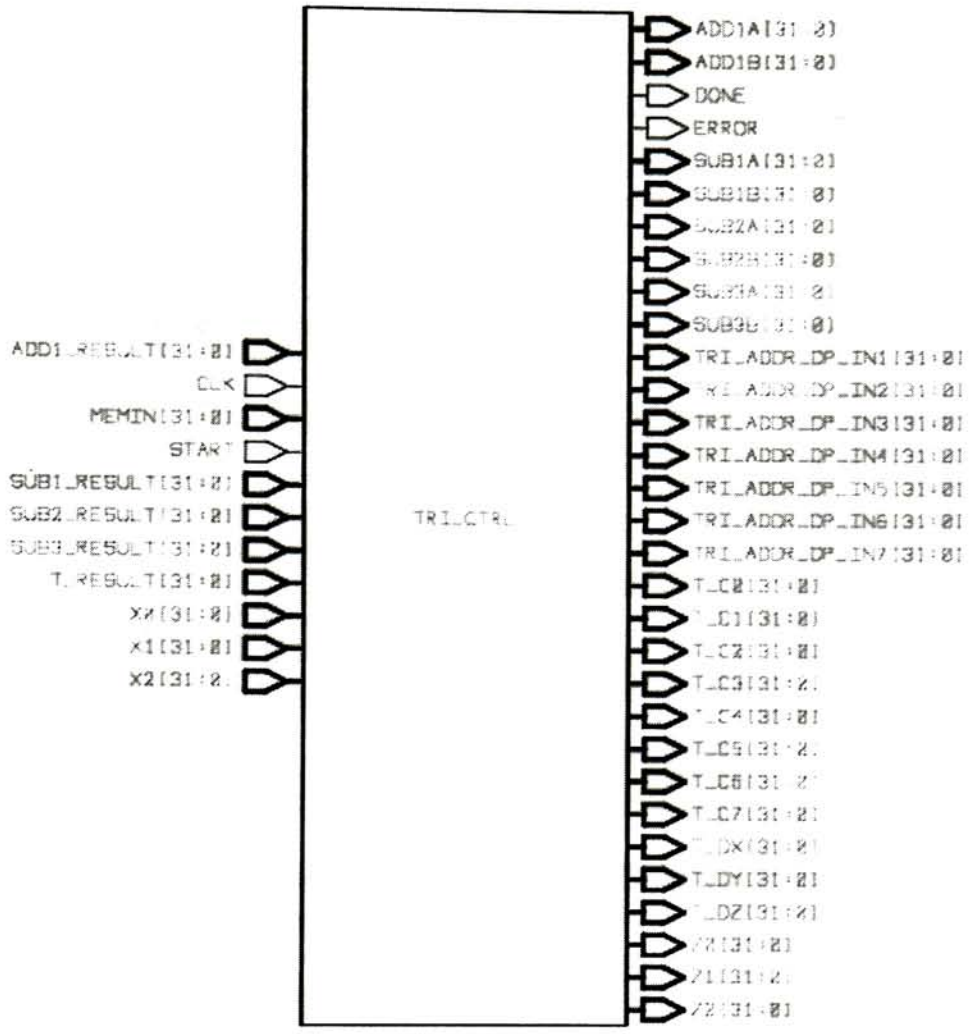


Figure 10: Top Level Diagram of Trilinear Interpolation TRI_CTRL Block

The area of this block is 18,148 cells. The maximum path delay through this block is 39.06 nanoseconds.

6.4.1.2.2 TRI_ADDR_BLOCK (tri_addr_block.vhd)

This block defines the datapath for calculating the memory address of a desired value in memory. The data needed from memory are the lattice points of the cell containing the input point and their corresponding points in the output. These lattice points are located by indices which were determined by searching the input space for the appropriate cell containing the input point. These indices are input to this block which computes the memory address of the desired values.

The top level diagram showing input and output signals of this block can be found in Figure 11. A low level block diagram is shown in Figure 12.

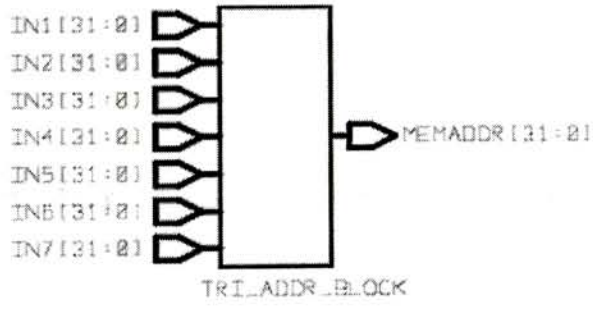


Figure 11: Top Level Diagram of Trilinear Interpolation TRI_ADDR_BLOCK

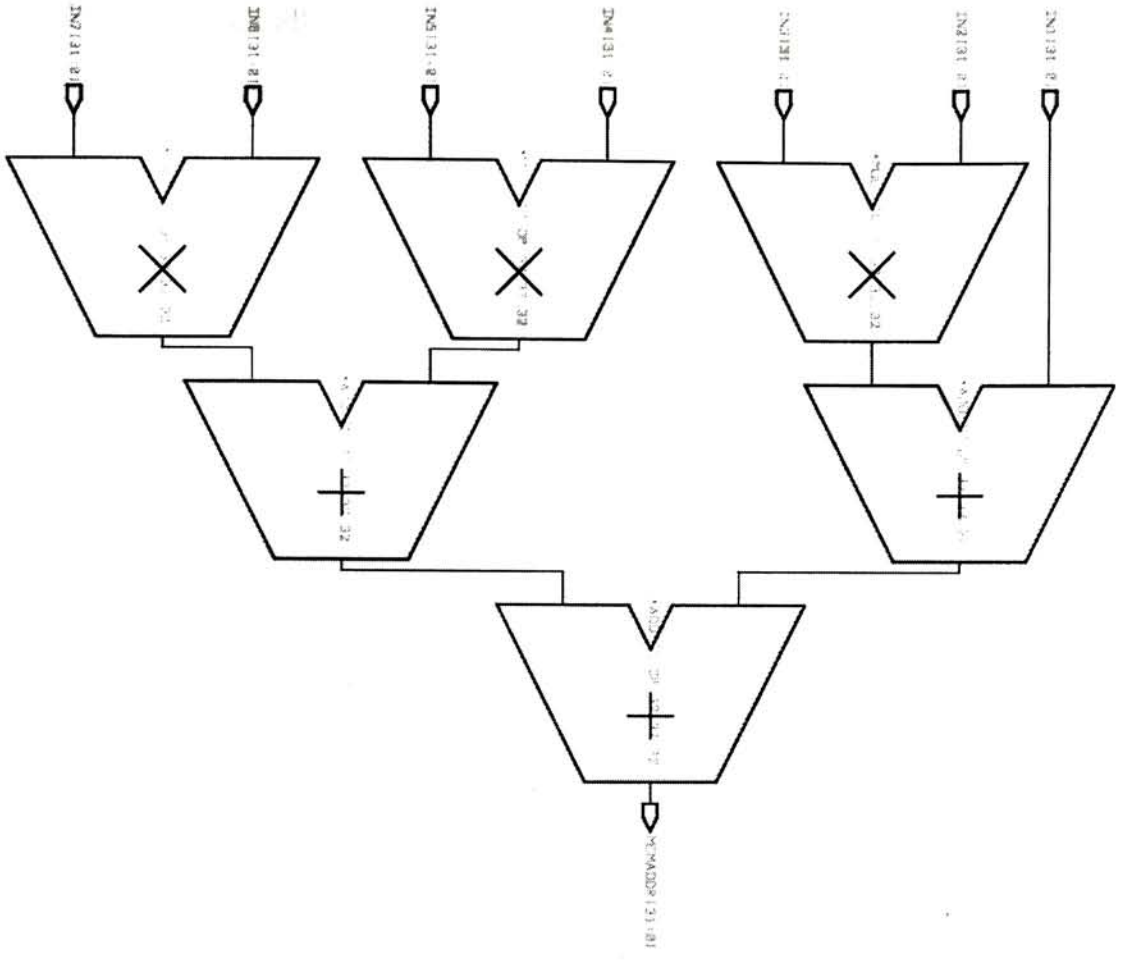


Figure 12: Datapath Architecture of TRI_ADDR_BLOCK

The area of this block is 20,014 cells. The maximum path delay through this block is 38.87 nanoseconds.

6.4.1.2.3 TRI_CX_BLOCK (tri_cx_block.vhd)

This block defines the datapath for calculating the C1 through C7 values (henceforth known as CX values) used to calculate the output. These values are calculated from the lattice points of the cell containing the input point. Only three subtractors and one adder

is needed to compute all seven values because these units are shared due to pipelining of operations by the control block.

The top level diagram showing input and output signals of this block can be found in Figure 13. A low level block diagram is shown in Figure 14.

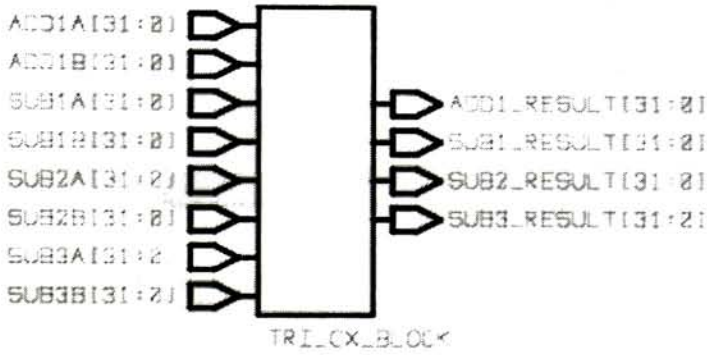


Figure 13: Top Level Diagram of Trilinear Interpolation TRI_CX_BLOCK

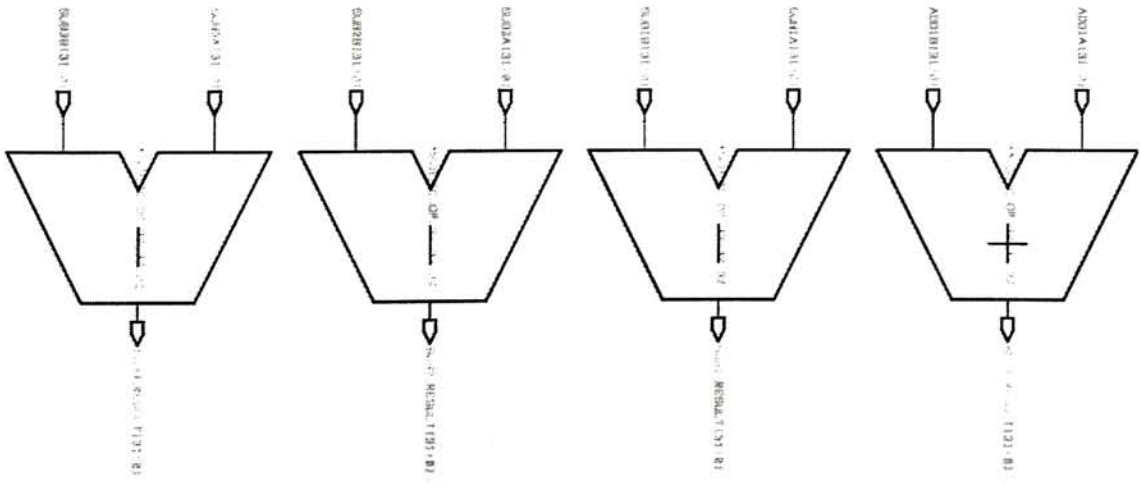


Figure 14: Datapath Architecture of TRI_CX_BLOCK

The area of this block is 3,658 cells. The maximum path delay through this block is 10.36 nanoseconds.

6.4.1.2.4 TRI_TERM_BLOCK (tri_term_block.vhd)

This block defines the datapath for calculating the interpolated values; it is the workhorse of the interpolation module. It uses the CX values produced by the TRI_CX_BLOCK and several other inputs provided by the control block to compute the interpolated values.

The top level diagram showing input and output signals of this block can be found in Figure 15. A low level block diagram is shown in Figure 16. The disadvantage of using scaled data can be seen in this low level diagram; there are divide units throughout the datapath that are needed to scale down the results of the multiplication-rich calculation performed by this block. These divides units add significant area and delays to this block.

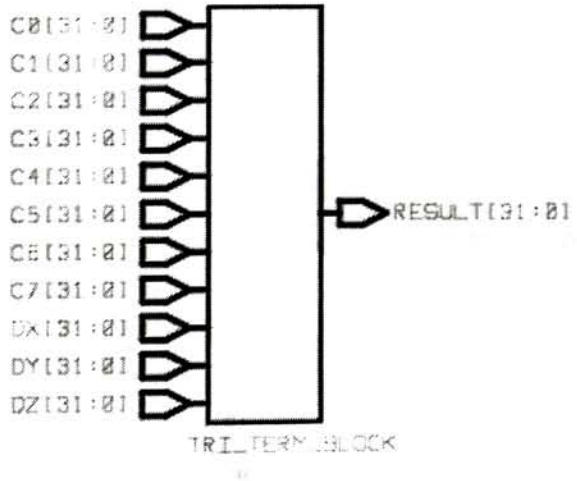


Figure 15: Top Level Diagram of Trilinear Interpolation TRI_TERM_BLOCK

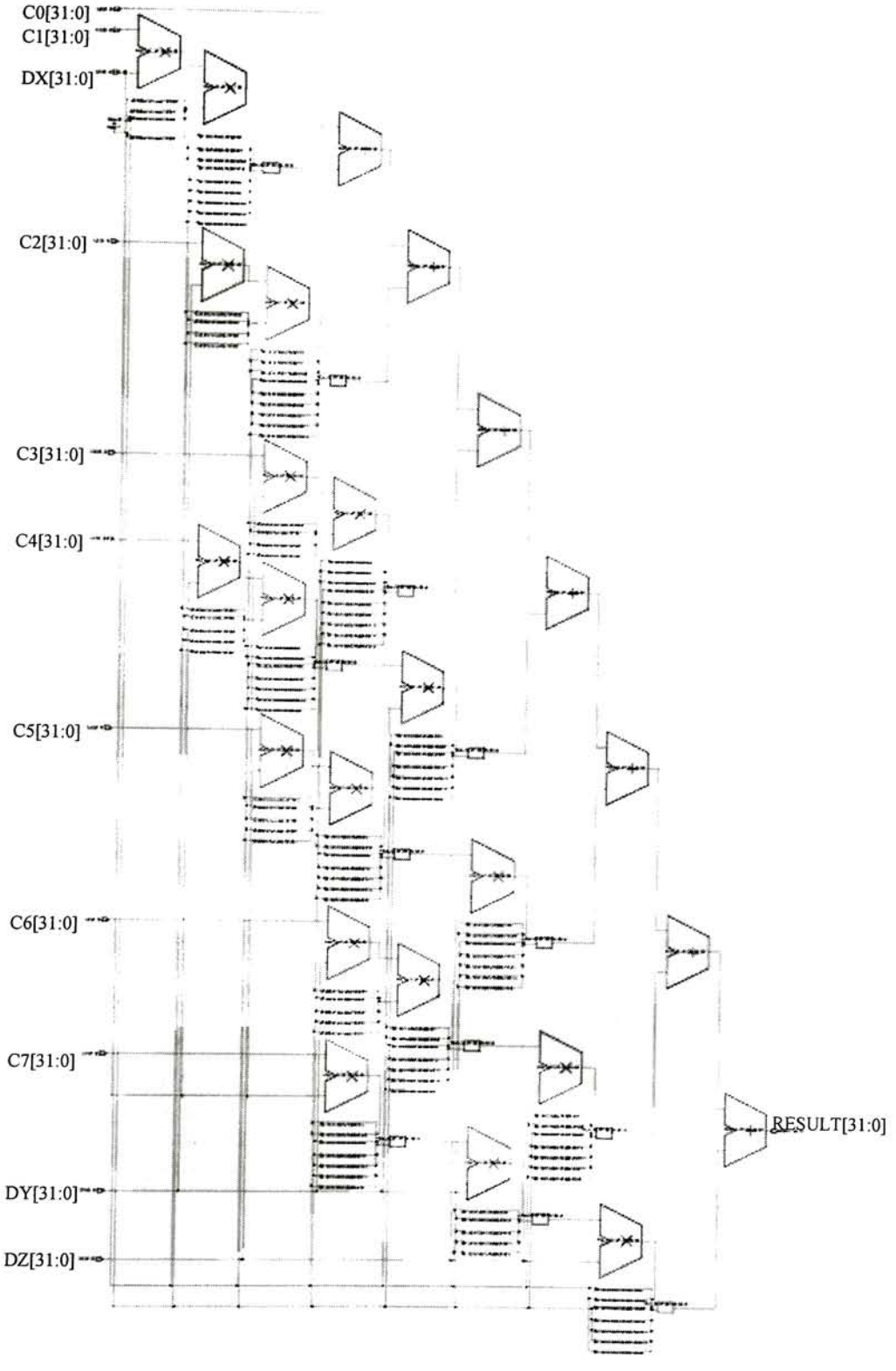


Figure 16: Datapath Architecture of TRI_TERM_BLOCK

The area of this block is 148,537 cells. The maximum path delay through this block is 445.6 nanoseconds.

6.4.1.3 Summary of Results

It is important to note that the multiply units in the sub-block designs operate on 32-bit integer input; the 64-bit result is truncated to the 32 most significant bits, which can result in truncation error.

Table 6 summarizes the area and timing results of the synthesized blocks for the trilinear interpolation module. Note that TRI_TERM_BLOCK requires multiple clock cycles to produce valid output. During this time, the control block will be exploiting parallelism by continuing with other processing, if possible. If not, it will simply wait the required number of clock cycles for valid data.

Table 6: Trilinear Interpolation Module Synthesis Results

Block	Area (cells)	Timing (ns)
TRI_CTRL_BLOCK	18,148	39.06
TRI_ADDR_BLOCK	20,014	38.87
TRI_CX_BLOCK	3,658	10.36
TRI_TERM_BLOCK	148,537	445.6
Total	190,357	--

6.4.2 ICI Module (ici_top.vhd)

Figure 17 shows the I/O signals of the ICI module; Figure 18 shows the block diagram of the module. Table 7 describes the input and output.

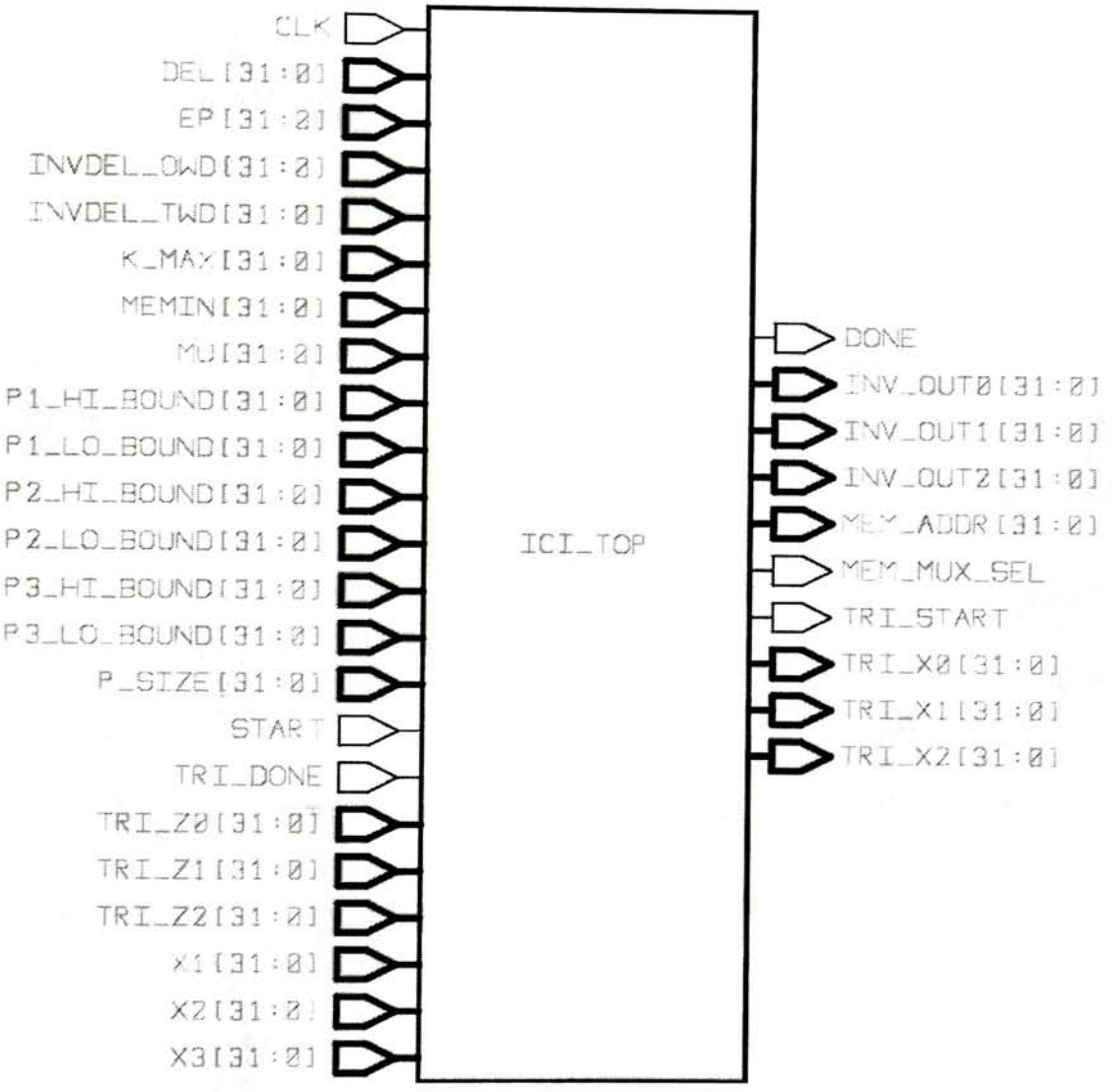


Figure 17: Top Level Diagram of ICI Module

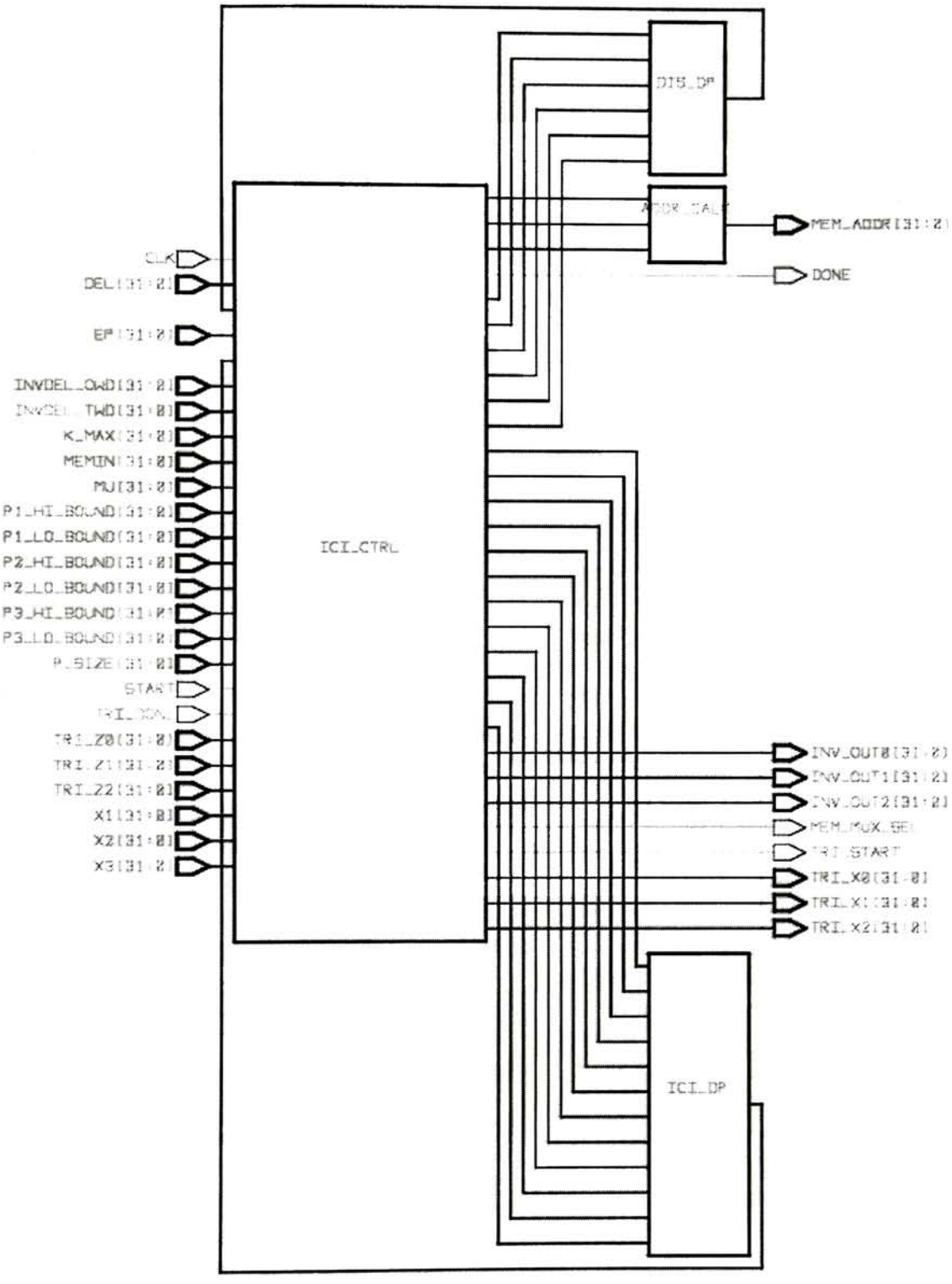


Figure 18: Submodule Diagram of ICI Module

Table 7: ICI Module I/O Signals

Signal Name	I/O	Type	Description
Clk	Input	Std_logic	System clock.
Start	Input	Std_logic	ICI operation begins on the rising edge of this signal.
Memin	Input	std_logic_vector[31:0]	Input data from external memory.
p_size	Input	std_logic_vector[31:0]	Specifies the number of rows in the LUT, where each row contains the three components of a point in the input space and the three components of the corresponding point in the output space.
X1	Input	std_logic_vector[31:0]	Specifies the first component of the input point.
X2	Input	std_logic_vector[31:0]	Specifics the second component of the input point.
X3	Input	std_logic_vector[31:0]	Specifies the third component of the input point.
Del	Input	std_logic_vector[31:0]	Specifies the delta amount to use for calculating gradient matrix derivatives.
Invdel_owd	Input	std_logic_vector[31:0]	Specifies the multiplier constant to use when calculating a forward OR backward gradient matrix derivative. Note: Must be equal to 10,000/del
Invdel_twd	Input	std_logic_vector[31:0]	Specifies the multiplier constant to use when calculating a forward AND backward gradient matrix derivative. Note: Must be equal to 10,000/2*del
K_max*	Input	std_logic_vector[31:0]	Specifies the maximum number of iterations to stop ICI iteration.
Ep	Input	std_logic_vector[31:0]	Specifies the threshold to stop ICI iteration.
Mu	Input	std_logic_vector[31:0]	Specifies the value for μ used in the ICI update equation.
P1_lo_bound**	Input	std_logic_vector[31:0]	Specifies the lower bound of the first component of the input space.
P1_hi_bound	Input	std_logic_vector[31:0]	Specifies the upper bound of the first component of the input space.
P2_lo_bound	Input	std_logic_vector[31:0]	Specifies the lower bound of the second component of the input space.
P2_hi_bound	Input	std_logic_vector[31:0]	Specifies the upper bound of the second component of the input space.
P3_lo_bound	Input	std_logic_vector[31:0]	Specifies the lower bound of the third component of the input space.

P3_hi_bound	Input	Std_logic_vector[31:0]	Specifies the upper bound of the third component of the input space.
Tri_z0	Input	Std_logic_vector[31:0]	First component of an interpolated value from the trilinear interpolation module.
Tri_z1	Input	Std_logic_vector[31:0]	Second component of an interpolated value from the trilinear interpolation module.
Tri_z2	Input	Std_logic_vector[31:0]	Third component of an interpolated value from the trilinear interpolation module.
Tri_done	Input	Std_logic	Signal from the trilinear interpolation module indicating interpolation is complete.
Done	Output	Std_logic	Signals the completion of the ICI operation.
Memaddr	Output	Std_logic_vector[31:0]	Specifies the location of the desired data in memory.
Inv_out0	Output	Std_logic_vector[31:0]	First component of the output value; valid when done=1.
Inv_out1	Output	Std_logic_vector[31:0]	Second component of the output value; valid when done=1.
Inv_out2	Output	Std_logic_vector[31:0]	Third component of the output value; valid when done=1.
Tri_start	Output	Std_logic	Signals the trilinear interpolation module to begin operation.
Tri_x0	Output	Std_logic_vector[31:0]	First component of the input value to the trilinear interpolation module.
Tri_x1	Output	Std_logic_vector[31:0]	Second component of the input value to the trilinear interpolation module.
Tri_x2	Output	Std_logic_vector[31:0]	Third component of the input value to the trilinear interpolation module.

* Unlike the other input parameters to this module, **k_max** is NOT a scaled value. **K_max** specifies the maximum number of iterations used to compute the inverse.

** For example, if the input colorspace was Lab:

P1_lo_bound = 0	and	P1_hi_bound=100	because $0 < L < 100$
P2_lo_bound = -127	and	P2_hi_bound=+127	because $-127 < a < +127$
P3_lo_bound = -127	and	P3_hi_bound=+127	because $-127 < b < +127$

6.4.2.1 Operation

The system clock must have a period of 40 nanoseconds or greater; this corresponds to a maximum operating frequency of 25 MHz. Simulation results show that this design can calculate the inverse for a given input point in 1.3 ms when operating at 25 MHz. A higher maximum operating frequency could be achieved if the design was targeted to a vendor-specific sub-micron technology library. This would reduce the time required to compute the inverse.

The rising edge of start begins ICI computation of input point specified by signals **X1**, **X2** and **X3**. **Done** is asserted when computation is complete. The output components occur on signals **inv_out0**, **inv_out1** and **inv_out2**. **Memin** and **memaddr** are interface signals to the external memory module. **Memin** is data coming from the memory; **memaddr** is the location in memory of desired data and is controlled by the interpolation module's control block.

6.4.2.2 Input Description

The inverse printer map has an input space defined by equally subdividing the output space of P. This creates a P^{-1} with a structured input which allows for fast and efficient interpolation. The drawback to this approach is that the input space will contain points that are outside of the gamut defined by P. These points which are outside of the gamut typically will not have an inverse, which results in a poor P^{-1}.

In order to obtain a better inverse map, points which are outside of the gamut are first mapped onto the gamut boundary. These "mapped" values will typically have an inverse, which result in a much better approximation of the inverse. When using ICI to compute the inverse, the "mapped" input points should be used. A more detail discussion of gamut mapping is found in Section 2.3.

6.4.2.3 Sub-blocks

6.4.2.3.1 ICI_CTRL (ici_ctrl.vhd)

This block contains the control logic needed to carry out the ICI computation of the inverse. Its input and output signals control data flow by interfacing to the other sub-blocks, to the external memory module, and to the trilinear interpolation module. It is also responsible for beginning the inverse computation when start is raised and for asserting done when complete.

The top level diagram is shown in Figure 19. A lower level block diagram is not available because it contains too many low-level components (primarily registers) and interconnects to be displayed on paper.

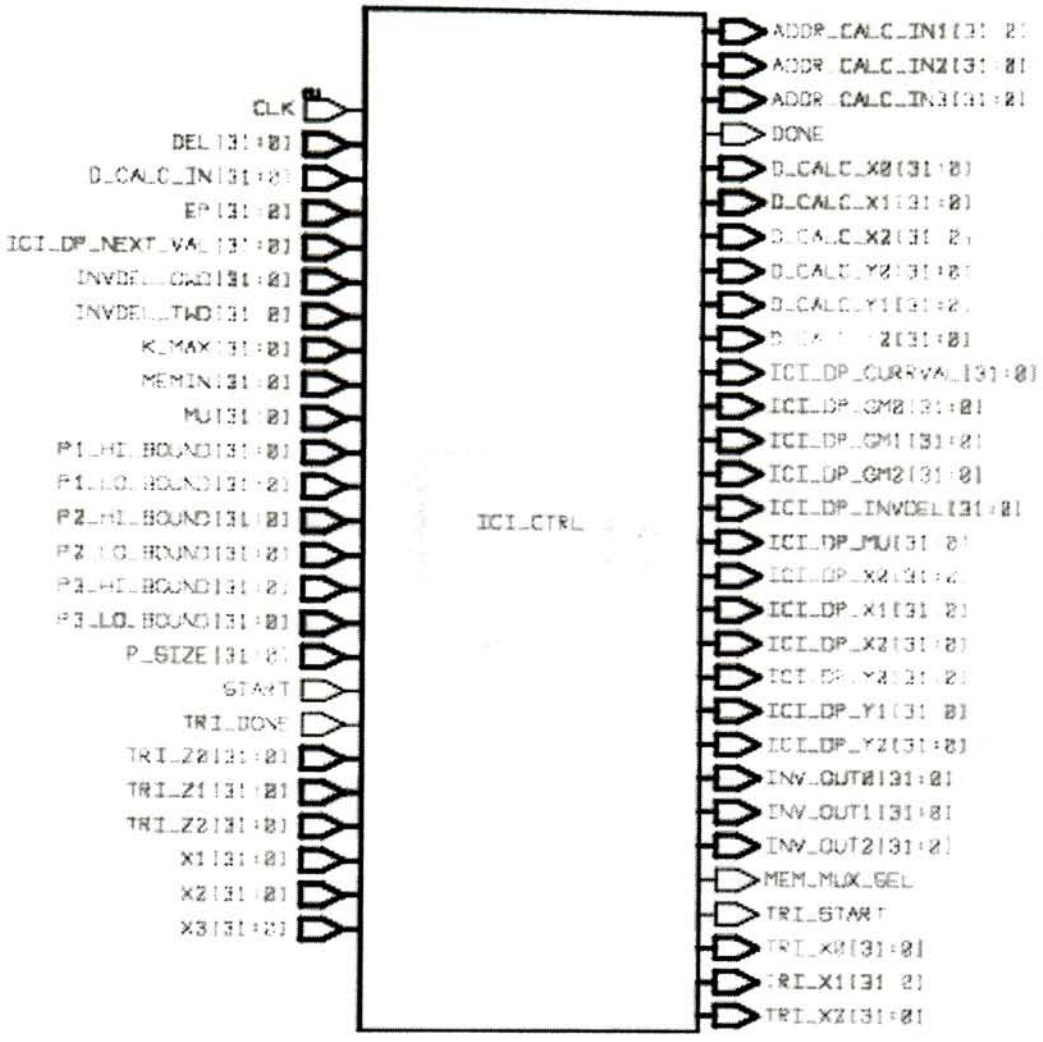


Figure 19: Top Level Diagram of ICI_CTRL Block

The area of this block is 43,359 cells. The maximum path delay through this block is 39.20 nanoseconds.

6.4.2.3.2 ADDR_CALC (addr_calc.vhd)

This block defines the datapath for calculating the memory address of a desired value in memory. Two values provided to this datapath are the index for a given entry in memory

and the offset of the value in that entry. These values are controlled by the ICI control block.

The top level diagram for this block is shown in Figure 20. The lower level block diagram is shown in Figure 21.

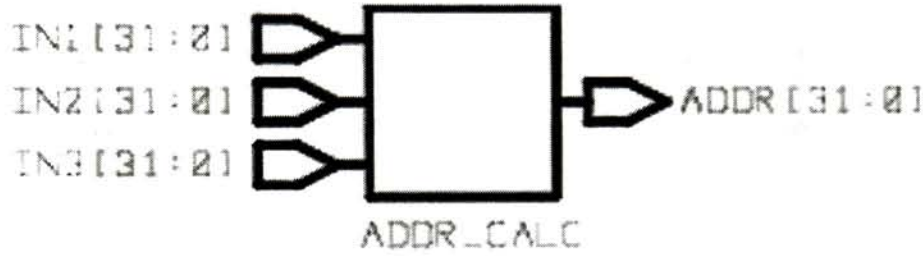


Figure 20: Top Level Diagram of ICI ADDR_CALC Block

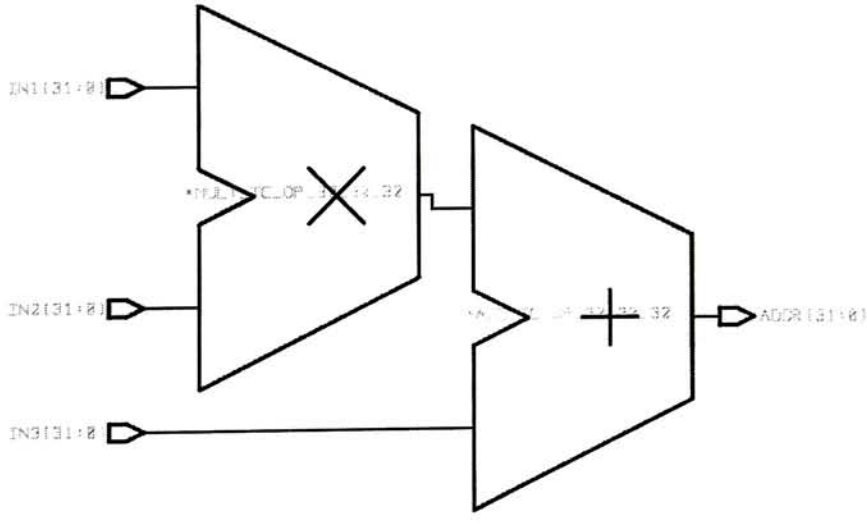


Figure 21: Datapath Architecture of ICI ADDR_CALC Block

The area of the block is 6,608 cells. The maximum path delay through this block is 31.37 nanoseconds.

6.4.2.3.3 DIS_DP (dis_dp.vhd)

This block defines the datapath needed to compute the Euclidean distance between two points. The distance between two points is required at several points in the ICI algorithm such as when computing the initial estimate and when computing the difference between current and previous computed output values to determine when to stop iterating.

The top level diagram for this block is shown in Figure 22. The lower level block diagram is shown in Figure 23.

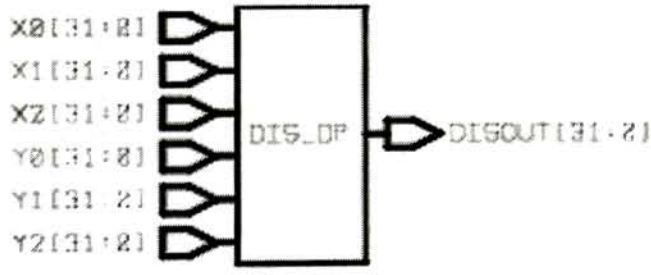


Figure 22: Top Level Diagram of ICI DIS_DP Block

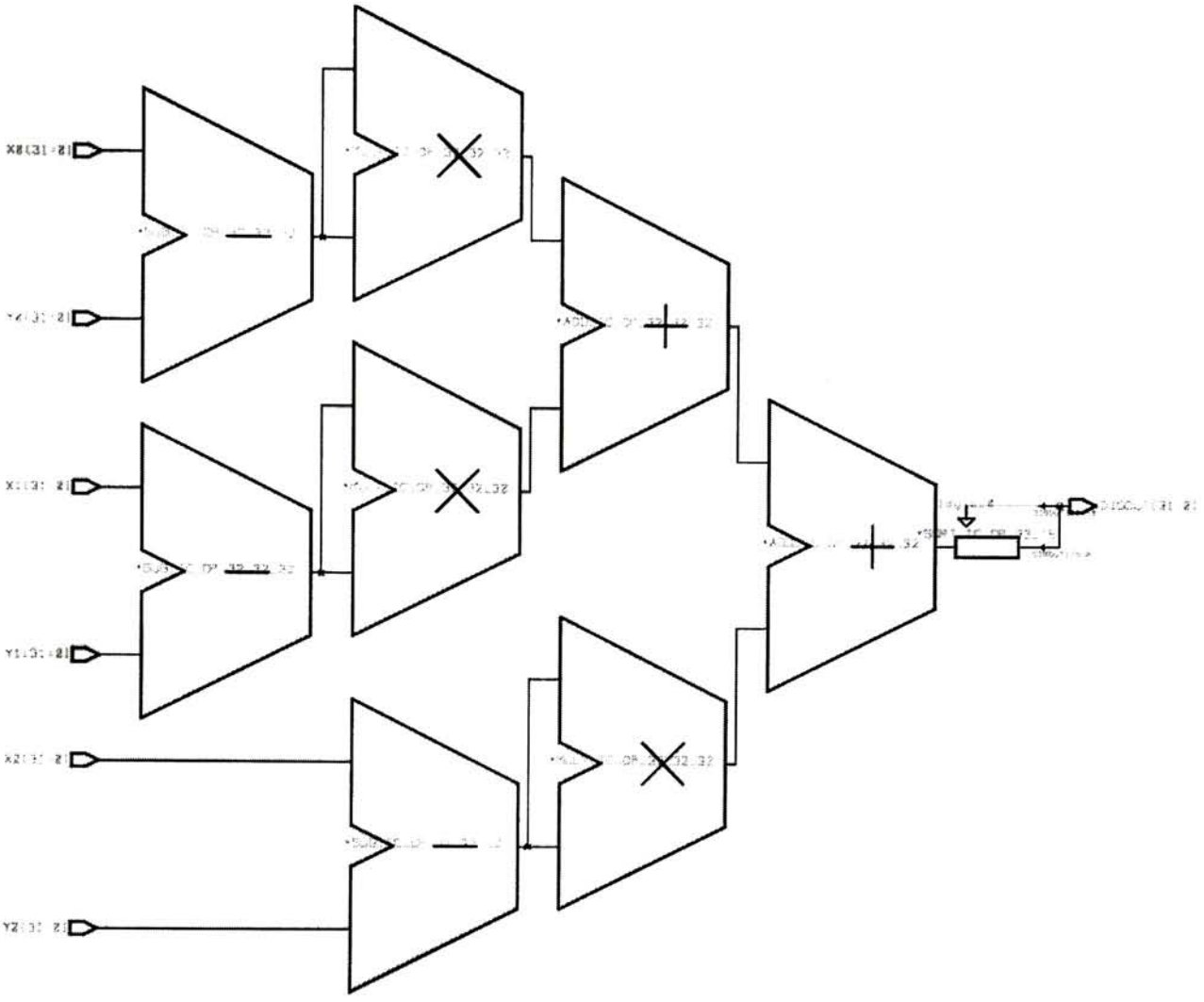


Figure 23: Datapath Architecture of ICI DIS_DP Block

The area of the block is 23,122 cells. The maximum path delay through this block is 183.46 nanoseconds.

6.4.2.3.4 ICI_DP (ici_dp.vhd)

This block defines the datapath that implements the update equation of the ICI algorithm. Inputs to this block are controlled by the ICI control block.

The top level diagram for this block is shown in Figure 24. The lower level block diagram is shown in Figure 25.

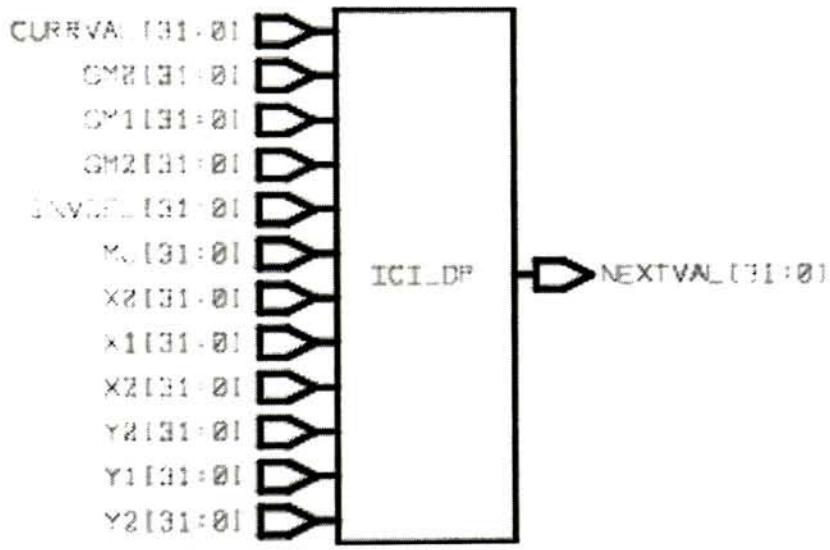


Figure 24: Top Level Digram of ICI_DP Block

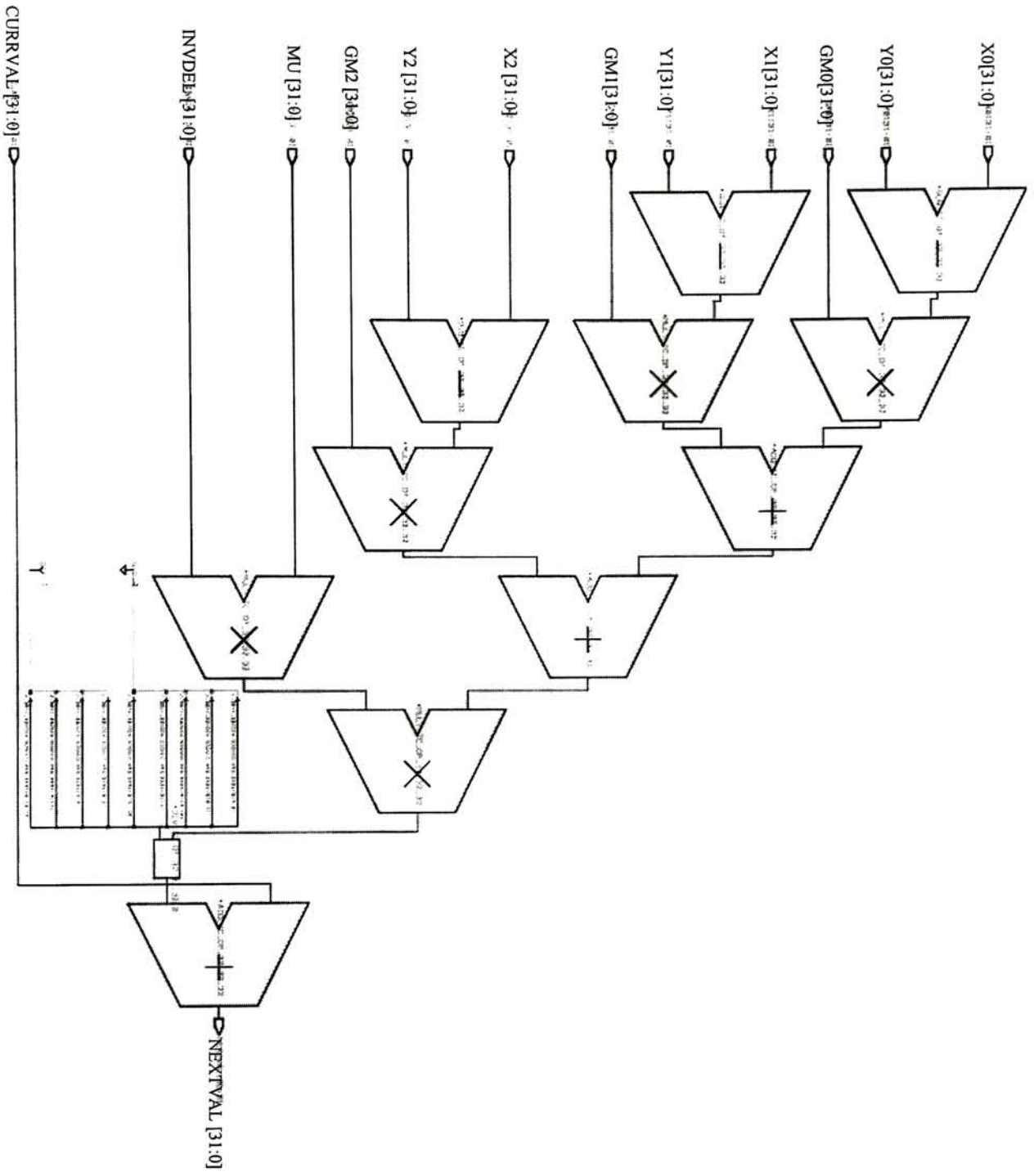


Figure 25: Datapath Architecture of ICI_DP Block

The area of the block is 36,469 cells. The maximum path delay through this block is 159.96 nanoseconds.

6.4.3 Summary of Results

Again, note that the multiply units in the sub-block designs operate on 32-bit integer input; the 64-bit result is truncated to the 32 most significant bits, which can result in truncation error.

Table 8 summarizes the area and timing results of the synthesized blocks for the ICI module. Note that the DIS_DP and ICI_DP blocks require multiple clock cycles to produce valid output. During this time, the control block will be exploiting parallelism by continuing with other processing, if possible. If not, it will simply wait the required number of clock cycles for valid data.

Table 8: ICI Module Synthesis Results

Block	Area (cells)	Timing (ns)
ICI_CTRL	43,359	39.20
ADDR_CALC	6,608	31.37
DIS_DP	23,122	183.46
ICI_DP	36,469	159.96
Total	112,683	--

6.5 System Overview

Figure 26 illustrates how the interpolation and ICI modules would be used in a real system; the arrows represent communication channels. As discussed previously, an external memory module would contain the data defining the P look-up table. Communication channels would exist between the interpolation, ICI, and external memory modules. Operating at the maximum frequency of 25 MHz, the system can construct the inverse printer map P^{-1} for a 13^3 forward map P in approximately 2.9

seconds (13^3*1.3ms). The computation time could be reduced if the modules operated with a higher clock frequency. This could be achieved by synthesizing this design to a vendor-specific sub-micron technology library.

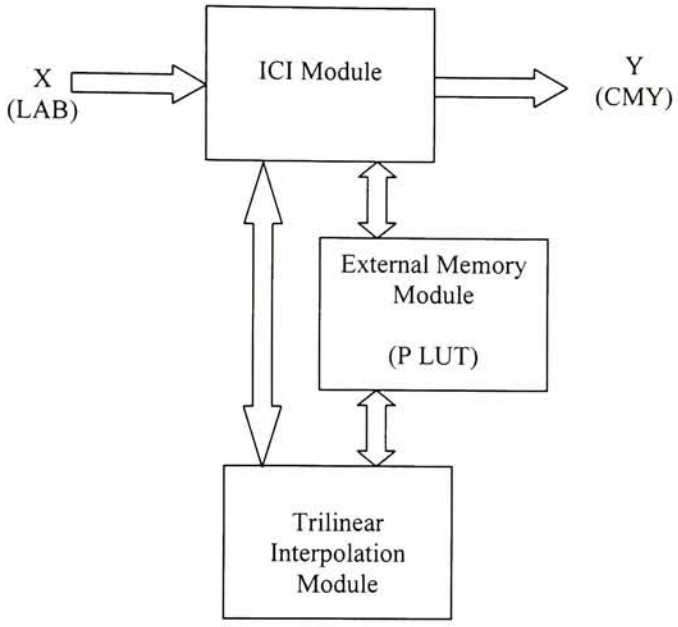


Figure 26: Diagram of System to Compute P^{-1}

Also note that the trilinear interpolation module can be used alone with an external memory containing an existing P^{-1} as illustrated in Figure 27. This configuration would provide a method of interpolating image data through P^{-1}. Operating at the maximum frequency of 25 MHz, this system could interpolate a 512x512 image in approximately 0.66 seconds (512*512*2.5μs). Again, this computation time could be reduced if the design was synthesized with a sub-micron technology library.

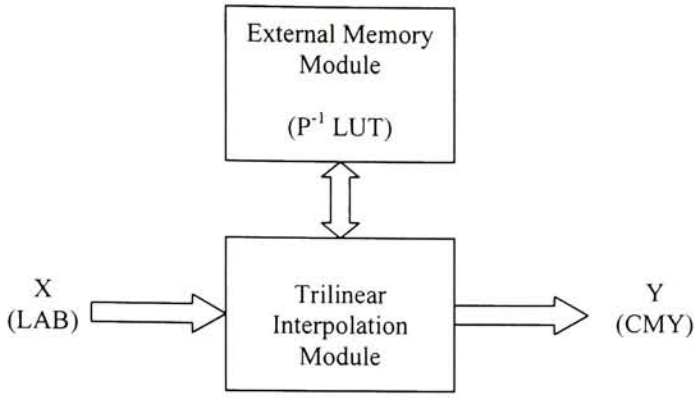


Figure 27: Diagram of System to Interpolate Image Data

7 Concluding Remarks

7.1 Conclusions

In this thesis project, three algorithms that compute the inverse of a forward printer map have been benchmarked. The algorithms under investigation were the Shepard's Interpolation, Moving Matrix, and Iteratively Clustered Interpolation (ICI) algorithms. The software simulations showed that the ICI algorithm was clearly the best algorithm in terms of accuracy and speed. A review of the algorithm and an initial hardware architecture to implement the ICI algorithm showed that this algorithm's complexity does not far exceed the complexity of the other two algorithms.

The ICI algorithm was implemented in VHDL and synthesized to a generic library. The area and timing results of the synthesized design showed that this algorithm can be implemented in hardware to target an application specific integrated circuit (ASIC). The VHDL source code and synthesized netlists provide a working design. Synthesizing to

the Synopsys generic library resulted in an ICI module containing 112,683 cells and a trilinear interpolation module containing 190,357 cells. The timing of the modules resulted in a maximum operating frequency of 25 MHz, corresponding to a 40 nanosecond clock period. Higher frequency of operation could be achieved if the design was synthesized to a deep sub-micron CMOS technology library.

7.2 Future Work

The future work of this project would take the hardware implementation further. The design can be mapped to a specific and current target technology library. Next, place and route would be performed for the design. Then, a real ASIC could be fabricated.

Ultimately, the ASIC would be embedded into a printer's electronics. A printer with an inline sensor that measures output color, along with this embedded ASIC, would provide the major components needed for an automatic and real-time color correction system.

7.3 Acknowledgements

I would like to thank my committee members, Drs. K. Hsu, S. Dianat, and A. Mathew, for their insights and guidance throughout this thesis work. I would also like to thank Dr. L. K. Mestha, our industry contact at Xerox, for his insight and guidance and for overseeing the research conducted during this project. This research was supported by a grant from the Center for Electronic Imaging Systems (CEIS), the New York State Office of Science, Technology and Academic Research (NYSTAR) and Xerox Corporation. A

big thanks to all of the aforementioned for everything they have taught me about color correction and the printing process and industry. Also thank you for the opportunity to work and study in such a challenging and timely area of research.

Lastly, I would like to thank Kishore Narreddy, a graduate student in the Electrical Engineering department at RIT, for his collaborative efforts in conducting this research. I would like to specifically thank Kishore for sharing his knowledge of gamut mapping and for providing the gamut mapping code. Also thanks to Kishore for his help in verifying the results of the software simulations.

8 References

[1] S. A. Dianat, L. K. Mestha, D. E. Viassolo, and Y. R. Wang. A practical algorithm for the inversion of an experimental input-output color map for color correction. October 2001.

[2] R. Balasubramanian, M. S. Maltz. Refinement of printer transformations using weighted regression. *SPIE Proceedings Vol. 2658*, 1996.

[3] D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. *Proc. ACM Nat. Conf., Pages 517--524*, 1968.

[4] M. W. Martin, K. K. Narredy. Analysis of Printer Inversion Models. April 2002. **UNPUBLISHED**.

[5] H. R. Kang. *Color Technology for Electronic Imaging Devices*. SPIE Press, 1997.

[6] R. Groff, P. Khargonekar, D. Koditschek, T. Thieret, and L. K. Mestha. Modeling and Control of Xerographic Processes. *Proceedings of the 38th Conference on Decision and Control*. December 1999.

[7] M. C. Stone, W. B. Cowan, J. C. Beatty. Color Gamut Mapping and the Printing of Digital Color Images. *ACM Transactions on Graphics, Vol. 7, No. 4*. October 1988.

[8] R. Groff, P. Khargonekar, D. Koditschek, and T. Thieret. Representation of Color Space Transformations for Effective Calibration and Control. *IS&T NIP16: 2000 International Conference on Digital Printing Technologies*, 2000.

[9] Po-Chieh Hung. Colorimetric calibration in electronic imaging devices using a look-up-table model and interpolations. *Journal of Electronic Imaging, Vol 2(1)*. January 1993.

[10] L. K. Mestha, Y. R. Wang, M. A. Scheuer, and T. E. Thieret. A Multilevel Modular Control Architecture for Image Reproduction. *Proceedings of the 1998 IEEE International Conference on Control Applications*. September 1998.

[11] E. K. Chong and S. H. Zak. *An Introduction to Optimization*. John Wiley & Sons Inc., New York, 1996.

9 Appendices

9.1 Software Simulation Accuracy Results

In this appendix, the accuracy results of each algorithm for varying parameters are presented. The tables show the accuracy statistics; from them the optimal parameter values are determined. The algorithms and their parameters are discussed in detail in Section 5.2.

9.1.1 Shepard's Interpolation

P	MU	Mean Delta E	Std Dev	Mean+2 *Std	Min	Max
1	1	33.55	10.97	55.50	2.24	78.17
1	2	17.61	6.89	31.38	0.64	47.32
1	3	5.20	3.00	11.20	0.15	19.76
1	4	1.97	0.93	3.82	0.12	7.63
1	5	1.72	0.60	2.92	0.15	3.64
1	6	1.82	0.65	3.13	0.15	4.04
2	1	33.30	10.97	55.24	2.18	77.57
2	2	17.36	6.97	31.29	0.71	49.18
2	3	5.00	2.91	10.82	0.15	18.31
2	4	1.86	0.80	3.47	0.15	6.55
2	**5**	**1.64**	**0.54**	**2.73**	**0.15**	**3.81**
2	6	1.76	0.60	2.97	0.13	3.27
3	1	33.32	10.98	55.28	2.16	77.19
3	2	17.56	7.06	31.68	0.70	50.06
3	3	5.17	3.02	11.21	0.15	18.01
3	4	1.90	0.84	3.57	0.15	6.54
3	5	1.65	0.55	2.74	0.15	4.01
3	6	1.76	0.60	2.97	0.12	3.68
4	1	33.11	10.92	54.95	2.14	76.59
4	2	17.18	6.91	31.14	0.60	49.22
4	3	4.95	2.92	10.79	0.15	17.25
4	4	1.87	0.81	3.49	0.15	6.23
4	5	1.66	0.56	2.78	0.15	4.06
4	6	1.78	0.62	3.02	0.15	3.86
5	1	33.07	10.91	54.88	2.13	76.35
5	2	17.14	6.98	31.10	0.59	49.08
5	3	4.95	10.81	10.81	0.15	17.54
5	4	1.88	0.82	3.52	0.15	6.49
5	5	1.67	0.57	2.81	0.13	4.11
5	6	1.79	0.63	3.05	0.15	3.97
6	1	33.10	10.91	54.92	2.12	76.27
6	2	17.25	7.01	31.26	0.61	49.25
6	3	5.03	2.98	10.99	0.15	18.01
6	4	1.90	0.84	3.58	0.15	6.76
6	5	1.67	0.58	2.83	0.15	4.15
6	6	1.79	0.64	3.07	0.15	4.04

9.1.2 Moving Matrix

MU	Mean	Std Dev	Mean +2*StdDev	Min	Max
1	17.45	9.65	36.75	0.29	52.08
2	10.36	6.27	22.9	0.05	36
3	3.87	2.68	9.23	0.06	15.66
4	1.61	2.53	6.66	0.03	77.74
5	**1.28**	**2.29**	**5.86**	**0.02**	**54.88**
6	1.85	5.72	13.28	0.03	77.74
7	3.1	9.24	21.58	0.01	88.25

9.1.3 ICI

EP = 0.5 (Constant)
KMAX = 50 (Constant)

DEL	MU	Mean	Std Dev	Mean +2*StdDev	Min	Max
5	1	0.49	0.14	0.76	0.05	2.01
5	2	0.45	0.10	0.66	0.05	1.85
5	3	0.43	0.11	0.65	0.05	1.68
5	4	0.43	0.43	1.30	0.05	14.57
5	5	0.57	2.00	4.56	0.03	47.42
5	8	4.23	12.93	30.10	0.05	91.03
5	10	10.37	20.65	51.66	0.05	104.76
7	1	0.50	0.13	0.75	0.05	1.97
7	2	0.46	0.09	0.65	0.05	1.68
7	3	0.44	0.10	0.64	0.05	1.68
7	4	0.43	0.40	1.23	0.05	15.92
7	5	0.58	1.98	4.54	0.04	47.11
7	8	4.65	13.56	31.77	0.04	92.51
7	10	11.06	21.21	53.49	0.02	103.13
10	1	0.50	0.13	0.75	0.08	1.95
10	2	0.46	0.09	0.64	0.07	1.68
10	3	0.44	0.10	0.63	0.08	1.68
10	4	0.45	0.50	1.45	0.06	14.57
10	5	0.59	2.00	4.59	0.03	43.57
10	8	4.48	13.31	31.09	0.05	92.40
10	10	11.23	21.28	53.79	0.02	104.69
12	1	0.50	0.13	0.77	0.10	1.97
12	2	0.46	0.09	0.64	0.10	1.68
12	3	0.44	0.10	0.63	0.10	1.68
12	4	0.44	0.41	1.26	0.07	13.19
12	5	0.57	1.84	4.26	0.04	41.53
12	8	4.21	12.94	30.08	0.02	89.18
12	10	10.58	20.71	52.00	0.03	101.85

MU = 4
DEL = 10

KMAX	EP	Mean	Std Dev	Mean +2*StdDev	Min	Max
25	0.5	0.45	0.46	1.37	0.06	13.73
25	0.6	0.52	0.46	1.44	0.06	13.73
25	0.7	0.59	0.47	1.52	0.06	13.73
25	0.8	0.65	0.47	1.59	0.08	13.73
25	1	0.78	0.48	1.74	0.08	13.73
25	1.3	0.94	0.51	1.97	0.08	13.73
25	1.5	1.05	0.54	2.13	0.08	13.73
50	0.5	0.45	0.50	1.45	0.06	14.57
50	0.6	0.52	0.50	1.52	0.06	14.57
50	0.7	0.59	0.50	1.60	0.06	14.57
50	0.8	0.65	0.51	1.67	0.08	14.57
50	1	0.78	0.52	1.82	0.08	14.57
50	1.3	0.95	0.55	2.04	0.08	14.57
50	1.5	1.05	0.57	2.20	0.08	14.57
75	0.5	0.44	0.46	1.37	0.06	13.90
75	0.6	0.51	0.46	1.44	0.06	13.90
75	0.7	0.59	0.46	1.52	0.06	13.90
75	0.8	0.65	0.47	1.59	0.08	13.90
75	1	0.78	0.48	1.74	0.08	13.90
75	1.3	0.94	0.51	1.97	0.08	13.90
75	1.5	1.05	0.54	2.13	0.08	13.90
100	0.5	0.45	0.50	1.45	0.06	14.57
100	0.6	0.52	0.50	1.52	0.06	14.57
100	0.7	0.59	0.50	1.60	0.06	14.57
100	0.8	0.65	0.51	1.67	0.08	14.57
100	1	0.78	0.52	1.82	0.08	14.57
100	1.3	0.95	0.55	2.04	0.08	14.57
100	1.5	1.05	0.57	2.20	0.08	14.57

9.2 VHDL Code

9.2.1 tri_top.vhd

```
-- Description:  Trilinear Module (TOP)
-- Filename: tri_top.vhd

library IEEE,DWARE;
use IEEE.numeric_std.all;
use DWARE.DW_Foundation.all;
use IEEE.STD_LOGIC_1164.all;

entity TRI_TOP is

   port (CLK, START: in STD_LOGIC;
        X0, X1, X2: in INTEGER;
        DONE: out STD_LOGIC;
        ERROR: out STD_LOGIC;
        Z0, Z1, Z2: out INTEGER;

        -- External Memory Control Signals
        MEMIN: in INTEGER;
        MEMADDR: out INTEGER);

end TRI_TOP;

architecture STRUCTURAL of TRI_TOP is

   component TRI_ADDR_BLOCK
      port (IN1, IN2, IN3, IN4, IN5, IN6, IN7: in INTEGER;
           MEMADDR: out INTEGER);
   end component;

   component TRI_CX_BLOCK
      port (SUB1A, SUB1B, SUB2A, SUB2B, SUB3A, SUB3B, ADD1A, ADD1B: in
INTEGER;
           SUB1_RESULT, SUB2_RESULT, SUB3_RESULT, ADD1_RESULT: out
INTEGER);
   end component;

   component TRI_TERM_BLOCK
      port (C0, C1, C2, C3, C4, C5, C6, C7, DX, DY, DZ: in INTEGER;
           RESULT: out INTEGER);
   end component;

   component TRI_CTRL
      port (CLK, START: in STD_LOGIC;
           X0, X1, X2: in INTEGER;
           Z0, Z1, Z2: out INTEGER;
           DONE: out STD_LOGIC;
           ERROR: out STD_LOGIC;
           -- I/O Signals Trilinear DP
           TRI_ADDR_DP_IN1, TRI_ADDR_DP_IN2, TRI_ADDR_DP_IN3,
```

```
          TRI_ADDR_DP_IN4, TRI_ADDR_DP_IN5, TRI_ADDR_DP_IN6,
          TRI_ADDR_DP_IN7: out INTEGER;
            -- I/O Signals Memory
          MEMIN: in INTEGER;
          -- I/O Signals to CX DP
          SUB1A, SUB1B, SUB2A, SUB2B, SUB3A, SUB3B, ADD1A, ADD1B: out
INTEGER;
          SUB1_RESULT, SUB2_RESULT, SUB3_RESULT, ADD1_RESULT: in
INTEGER;
          -- I/O Signals to TERM DP
            T_C0, T_C1, T_C2, T_C3, T_C4, T_C5, T_C6, T_C7,
          T_DX, T_DY, T_DZ: out INTEGER;
          T_RESULT: in INTEGER);
   end component;

   -- Internal Signals for TRI_ADDR_BLOCK
   signal INT_IN1, INT_IN2, INT_IN3, INT_IN4,
        INT_IN5, INT_IN6, INT_IN7: INTEGER;

   -- Internal Signals for TRI_CX_BLOCK
   signal INT_SUB1A, INT_SUB1B, INT_SUB2A, INT_SUB2B, INT_SUB3A,
INT_SUB3B,
        INT_ADD1A, INT_ADD1B, INT_SUB1_RESULT, INT_SUB2_RESULT,
        INT_SUB3_RESULT,  INT_ADD1_RESULT: INTEGER;

   -- Internal Signals for TRI_TERM_BLOCK
   signal INT_C0, INT_C1,  INT_C2,  INT_C3,  INT_C4,  INT_C5,  INT_C6,
INT_C7,
          INT_DX,  INT_DY,  INT_DZ, INT_RESULT: INTEGER;

begin

   ADDR_BLK: TRI_ADDR_BLOCK
      port map (IN1=>INT_IN1,
                IN2=>INT_IN2,
                IN3=>INT_IN3,
                IN4=>INT_IN4,
                IN5=>INT_IN5,
                IN6=>INT_IN6,
                IN7=>INT_IN7,
               MEMADDR=>MEMADDR);

   CX_BLK: TRI_CX_BLOCK
      port map (SUB1A=>INT_SUB1A,
             SUB1B=>INT_SUB1B,
             SUB2A=>INT_SUB2A,
             SUB2B=>INT_SUB2B,
             SUB3A=>INT_SUB3A,
             SUB3B=>INT_SUB3B,
             ADD1A=>INT_ADD1A,
             ADD1B=>INT_ADD1B,
               SUB1_RESULT=>INT_SUB1_RESULT,
             SUB2_RESULT=>INT_SUB2_RESULT,
             SUB3_RESULT=>INT_SUB3_RESULT,
             ADD1_RESULT=>INT_ADD1_RESULT);

   TERM_BLK: TRI_TERM_BLOCK
```

```
   port map (C0=>INT_C0,
             C1=>INT_C1,
          C2=>INT_C2,
          C3=>INT_C3,
          C4=>INT_C4,
          C5=>INT_C5,
          C6=>INT_C6,
          C7=>INT_C7,
          DX=>INT_DX,
          DY=>INT_DY,
          DZ=>INT_DZ,
            RESULT=>INT_RESULT);

CTRL_BLK: TRI_CTRL
   port map (CLK=>CLK,
            START=>START,
            X0=>X0,
            X1=>X1,
            X2=>X2,
            Z0=>Z0,
            Z1=>Z1,
            Z2=>Z2,
            DONE=>DONE,
           ERROR=>ERROR,

            -- I/O Signals Trilinear DP
            TRI_ADDR_DP_IN1=>INT_IN1,
            TRI_ADDR_DP_IN2=>INT_IN2,
            TRI_ADDR_DP_IN3=>INT_IN3,
            TRI_ADDR_DP_IN4=>INT_IN4,
            TRI_ADDR_DP_IN5=>INT_IN5,
            TRI_ADDR_DP_IN6=>INT_IN6,
            TRI_ADDR_DP_IN7=>INT_IN7,

              -- I/O Signals Memory
            MEMIN=>MEMIN,

            -- I/O Signals to CX DP
            SUB1A=>INT_SUB1A,
            SUB1B=>INT_SUB1B,
            SUB2A=>INT_SUB2A,
            SUB2B=>INT_SUB2B,
            SUB3A=>INT_SUB3A,
            SUB3B=>INT_SUB3B,
            ADD1A=>INT_ADD1A,
            ADD1B=>INT_ADD1B,
            SUB1_RESULT=>INT_SUB1_RESULT,
            SUB2_RESULT=>INT_SUB2_RESULT,
            SUB3_RESULT=>INT_SUB3_RESULT,
            ADD1_RESULT=>INT_ADD1_RESULT,

            -- I/O Signals to TERM DP
              T_C0=>INT_C0,
              T_C1=>INT_C1,
            T_C2=>INT_C2,
            T_C3=>INT_C3,
            T_C4=>INT_C4,
```

```
                    T_C5=>INT_C5,
                    T_C6=>INT_C6,
                    T_C7=>INT_C7,
                    T_DX=>INT_DX,
                    T_DY=>INT_DY,
                    T_DZ=>INT_DZ,
                    T_RESULT=>INT_RESULT);

end STRUCTURAL;
```

9.2.2 tri_ctrl.vhd

```
-- Description:  Trilinear Control
-- Filename: tri_ctrl.vhd

library IEEE,DWARE;
use IEEE.numeric_std.all;
use DWARE.DW_Foundation.all;
--library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity TRI_CTRL is

   port (CLK, START: in STD_LOGIC;
        X0, X1, X2: in INTEGER;
        Z0, Z1, Z2: out INTEGER;
        DONE: out STD_LOGIC;
        ERROR: out STD_LOGIC;

        -- I/O Signals Trilinear DP
        TRI_ADDR_DP_IN1, TRI_ADDR_DP_IN2, TRI_ADDR_DP_IN3,
        TRI_ADDR_DP_IN4, TRI_ADDR_DP_IN5, TRI_ADDR_DP_IN6,
        TRI_ADDR_DP_IN7: out INTEGER;

          -- I/O Signals Memory
        MEMIN: in INTEGER;

        -- I/O Signals to CX DP
        SUB1A, SUB1B, SUB2A, SUB2B, SUB3A, SUB3B, ADD1A, ADD1B: out
INTEGER;
        SUB1_RESULT, SUB2_RESULT, SUB3_RESULT, ADD1_RESULT: in INTEGER;

        -- I/O Signals to TERM DP
          T_C0, T_C1, T_C2, T_C3, T_C4, T_C5, T_C6, T_C7,
        T_DX, T_DY, T_DZ: out INTEGER;
        T_RESULT: in INTEGER);

end TRI_CTRL;

architecture BEHAVIORAL of TRI_CTRL is
begin

   process
      variable I: INTEGER range 0 to 13;
      variable x_lo_pos_temp, y_lo_pos_temp, z_lo_pos_temp,
             x_lo_pos, y_lo_pos, z_lo_pos,
             x_hi_pos, y_hi_pos, z_hi_pos: INTEGER range 0 to 12;
      variable x_lo_temp, y_lo_temp, z_lo_temp,
             x_lo, x_hi, y_lo, y_hi, z_lo, z_hi: INTEGER;
      variable found_x, found_y, found_z: BOOLEAN;
      variable P001, P010, P011, P101: INTEGER;
      variable C0, C1, C2, C3, C4, C5, C6, C7: INTEGER;
      variable C4_TEMP, C5_TEMP, C6_TEMP, C7_TEMP1, C7_TEMP2: INTEGER;
      variable RESULT1, RESULT2, RESULT3: INTEGER;
```

```
begin

   loop
      TRI_ADDR_DP_IN1 <= 0;
    TRI_ADDR_DP_IN2 <= 0;
    TRI_ADDR_DP_IN3 <= 0;
    TRI_ADDR_DP_IN4 <= 0;
    TRI_ADDR_DP_IN5 <= 0;
    TRI_ADDR_DP_IN6 <= 0;
    TRI_ADDR_DP_IN7 <= 0;
    ERROR <= '0';

    wait until CLK'event and CLK='1';
      DONE <= '0';
    ERROR <= '0';

    if (START = '1') then

       found_x := FALSE;
       found_y := FALSE;
       found_z := FALSE;

       I:=0;
         TRI_ADDR_DP_IN1 <= 2;
       TRI_ADDR_DP_IN2 <= 6;
       TRI_ADDR_DP_IN3 <= I;
       TRI_ADDR_DP_IN4 <= 0;
       TRI_ADDR_DP_IN5 <= 0;
       TRI_ADDR_DP_IN6 <= 0;
       TRI_ADDR_DP_IN7 <= 0;
       wait until CLK'event and CLK='1';

         I:=I+1;
       TRI_ADDR_DP_IN1 <= 2;
       TRI_ADDR_DP_IN2 <= 6;
       TRI_ADDR_DP_IN3 <= I;
       wait until CLK'event and CLK='1';

         x_lo_pos_temp := I-1;
         x_lo_temp := MEMIN;
       y_lo_pos_temp := I-1;
             y_lo_temp := MEMIN;
       z_lo_pos_temp := I-1;
       z_lo_temp := MEMIN;
       I:=I+1;
       TRI_ADDR_DP_IN1 <= 2;
       TRI_ADDR_DP_IN2 <= 6;
       TRI_ADDR_DP_IN3 <= I;
       wait until CLK'event and CLK='1';

         while I <= 13 loop
          if X0 <= MEMIN and found_x = FALSE then  -- Read Mem 2
               x_lo_pos := x_lo_pos_temp;
            x_lo := x_lo_temp;
            x_hi_pos := I-1;
            x_hi := MEMIN;
```

```
      found_x := TRUE;
     else
      x_lo_pos_temp := I-1;
      x_lo_temp := MEMIN;
   end if;
   if X1 <= MEMIN and found_y = FALSE then
       y_lo_pos := y_lo_pos_temp;
      y_lo := y_lo_temp;
      y_hi_pos := I-1;
      y_hi := MEMIN;
      found_y := TRUE;
     else
      y_lo_pos_temp := I-1;
      y_lo_temp := MEMIN;
   end if;
   if X2 <= MEMIN and found_z = FALSE then
       z_lo_pos := z_lo_pos_temp;
      z_lo := z_lo_temp;
      z_hi_pos := I-1;
      z_hi := MEMIN;
      found_z := TRUE;
     else
      z_lo_pos_temp := I-1;
      z_lo_temp := MEMIN;
   end if;
   I:=I+1;
   TRI_ADDR_DP_IN1 <= 2;
   TRI_ADDR_DP_IN2 <= 6;
   TRI_ADDR_DP_IN3 <= I;
   wait until CLK'event and CLK='1';
end loop;

if found_x /= TRUE or found_y /= TRUE or found_z /= TRUE then
   ERROR <= '1';
  else

    I:=3;
   while I < 6 loop
       -- Get P000
      TRI_ADDR_DP_IN1 <= I;
      TRI_ADDR_DP_IN2 <= 1014;
      TRI_ADDR_DP_IN3 <= x_lo_pos;
      TRI_ADDR_DP_IN4 <= 78;
      TRI_ADDR_DP_IN5 <= y_lo_pos;
      TRI_ADDR_DP_IN6 <= 6;
      TRI_ADDR_DP_IN7 <= z_lo_pos;
      SUB1A <= X0;
      SUB1B <= x_lo;
      SUB2A <= X1;
      SUB2B <= y_lo;
      SUB3A <= X2;
      SUB3B <= z_lo;
      wait until CLK'event and CLK='1';

        -- Get P100
      TRI_ADDR_DP_IN1 <= I;
      TRI_ADDR_DP_IN2 <= 1014;
```

```
TRI_ADDR_DP_IN3 <= x_hi_pos;
TRI_ADDR_DP_IN4 <= 78;
TRI_ADDR_DP_IN5 <= y_lo_pos;
TRI_ADDR_DP_IN6 <= 6;
TRI_ADDR_DP_IN7 <= z_lo_pos;
T_DX <= SUB1_RESULT;
T_DY <= SUB2_RESULT;
T_DZ <= SUB3_RESULT;
wait until CLK'event and CLK='1';

  -- P000 Available
C0 := MEMIN;                    -- P000
SUB1B <= MEMIN;
  -- Get P010
TRI_ADDR_DP_IN1 <= I;
TRI_ADDR_DP_IN2 <= 1014;
TRI_ADDR_DP_IN3 <= x_lo_pos;
TRI_ADDR_DP_IN4 <= 78;
TRI_ADDR_DP_IN5 <= y_hi_pos;
TRI_ADDR_DP_IN6 <= 6;
TRI_ADDR_DP_IN7 <= z_lo_pos;
wait until CLK'event and CLK='1';

-- P100 Available
SUB1A <= MEMIN;              -- P100
  -- Get P001
TRI_ADDR_DP_IN1 <= I;
TRI_ADDR_DP_IN2 <= 1014;
TRI_ADDR_DP_IN3 <= x_lo_pos;
TRI_ADDR_DP_IN4 <= 78;
TRI_ADDR_DP_IN5 <= y_lo_pos;
TRI_ADDR_DP_IN6 <= 6;
TRI_ADDR_DP_IN7 <= z_hi_pos;
wait until CLK'event and CLK='1';

-- C1 Available
C1 := SUB1_RESULT;
  -- P010 Available
P010 := MEMIN;
SUB1A <= P010;
  -- Get P110
TRI_ADDR_DP_IN1 <= I;
TRI_ADDR_DP_IN2 <= 1014;
TRI_ADDR_DP_IN3 <= x_hi_pos;
TRI_ADDR_DP_IN4 <= 78;
TRI_ADDR_DP_IN5 <= y_hi_pos;
TRI_ADDR_DP_IN6 <= 6;
TRI_ADDR_DP_IN7 <= z_lo_pos;
wait until CLK'event and CLK='1';

-- C2 Available
C2 := SUB1_RESULT;
-- P001 Available
P001 := MEMIN;
SUB1A <= P001;
  -- Get P101
TRI_ADDR_DP_IN1 <= I;
```

```
TRI_ADDR_DP_IN2 <= 1014;
TRI_ADDR_DP_IN3 <= x_hi_pos;
TRI_ADDR_DP_IN4 <= 78;
TRI_ADDR_DP_IN5 <= y_lo_pos;
TRI_ADDR_DP_IN6 <= 6;
TRI_ADDR_DP_IN7 <= z_hi_pos;
wait until CLK'event and CLK='1';

  -- C3 Available
C3 := SUB1_RESULT;
-- P110 Avaiable
SUB1A <= MEMIN;         -- P110
SUB1B <= P010;
  -- Get P011
TRI_ADDR_DP_IN1 <= I;
TRI_ADDR_DP_IN2 <= 1014;
TRI_ADDR_DP_IN3 <= x_lo_pos;
TRI_ADDR_DP_IN4 <= 78;
TRI_ADDR_DP_IN5 <= y_hi_pos;
TRI_ADDR_DP_IN6 <= 6;
TRI_ADDR_DP_IN7 <= z_hi_pos;
wait until CLK'event and CLK='1';

  C4_TEMP := SUB1_RESULT;
SUB1A <= C4_TEMP;
SUB1B <= C1;
-- P101 Available
P101 := MEMIN;
SUB2A <= P101;
SUB2B <= P001;
  -- Get P111
TRI_ADDR_DP_IN1 <= I;
TRI_ADDR_DP_IN2 <= 1014;
TRI_ADDR_DP_IN3 <= x_hi_pos;
TRI_ADDR_DP_IN4 <= 78;
TRI_ADDR_DP_IN5 <= y_hi_pos;
TRI_ADDR_DP_IN6 <= 6;
TRI_ADDR_DP_IN7 <= z_hi_pos;
wait until CLK'event and CLK='1';

C4 := SUB1_RESULT;
  C5_TEMP := SUB2_RESULT;
SUB1A <= C5_TEMP;
SUB1B <= C1;
-- P011 Available
P011 := MEMIN;
SUB2A <= P011;
SUB2B <= P001;
wait until CLK'event and CLK='1';

C5 := SUB1_RESULT;
C6_TEMP := SUB2_RESULT;
SUB1A <= C6_TEMP;
SUB1B <= C2;
-- P111 Available
SUB2A <= MEMIN;         -- P111
SUB2B <= P011;
```

```
        SUB3A <= P001;
        SUB3B <= C4;
        wait until CLK'event and CLK='1';

          C6 := SUB1_RESULT;
        C7_TEMP1 := SUB2_RESULT;
        C7_TEMP2 := SUB3_RESULT;
        SUB1A <= C7_TEMP2;
        SUB1B <= P101;
        wait until CLK'event and CLK='1';

          ADD1A <= C7_TEMP1;
        ADD1B <= SUB1_RESULT;
        wait until CLK'event and CLK='1';

        C7 := ADD1_RESULT;

        T_C0 <= C0;
        T_C1 <= C1;
        T_C2 <= C2;
        T_C3 <= C3;
        T_C4 <= C4;
        T_C5 <= C5;
        T_C6 <= C6;
        T_C7 <= C7;

          if (I=4) then
           RESULT1 := T_RESULT;
          elsif (I=5) then
           RESULT2 := T_RESULT;
          end if;

         I:=I+1;
       end loop;

         -- Required wait cycles for term_block
      -- to finish computation of last
      -- component interpolation.
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';
         wait until CLK'event and CLK = '1';

      Z0 <= RESULT1;
      Z1 <= RESULT2;
      Z2 <= T_RESULT;
        DONE <= '1';
    end if;
end if;
```

```
        end loop;
    end process;

end BEHAVIORAL;
```

9.2.3 tri_cx_block.vhd

```
-- Description:  Trilinear CX Block
-- Filename: tri_cx_block.vhd

library IEEE,DWARE;
use IEEE.numeric_std.all;
use DWARE.DW_Foundation.all;
--library IEEE;
--use IEEE.STD_LOGIC_1164.all;

entity TRI_CX_BLOCK is

   port (SUB1A, SUB1B, SUB2A, SUB2B, SUB3A, SUB3B, ADD1A, ADD1B: in
INTEGER;
       SUB1_RESULT, SUB2_RESULT, SUB3_RESULT, ADD1_RESULT: out INTEGER);

end TRI_CX_BLOCK;

architecture BEHAVIORAL of TRI_CX_BLOCK is
begin

   SUB1_RESULT <= SUB1A - SUB1B;
   SUB2_RESULT <= SUB2A - SUB2B;
   SUB3_RESULT <= SUB3A - SUB3B;
   ADD1_RESULT <= ADD1A + ADD1B;

end BEHAVIORAL;
```

9.2.4 tri_term_block.vhd

```
-- Description:  Term Block
-- Filename: tri_term_block.vhd
-- Revisions: 4/13/02  Creation.

library IEEE, DWARE;
--library IEEE;
use IEEE.NUMERIC_STD.all;
use IEEE.STD_LOGIC_1164.all;
use DWARE.DW_Foundation.all;

entity TRI_TERM_BLOCK is

  port (C0, C1, C2, C3, C4, C5, C6, C7, DX, DY, DZ: in INTEGER;
      RESULT: out INTEGER);

end TRI_TERM_BLOCK;

architecture BEHAVIORAL of TRI_TERM_BLOCK is
   signal TERM1, TERM2, TERM3, TERM4, TERM5, TERM6, TERM7: INTEGER;
begin

   TERM1 <= TO_INTEGER(TO_SIGNED((C1*DX*47), 32) /
TO_SIGNED(100000,32));
   TERM2 <= TO_INTEGER(TO_SIGNED((C2*DY*47), 32) /
TO_SIGNED(100000,32));
   TERM3 <= TO_INTEGER(TO_SIGNED((C3*DZ*47), 32) /
TO_SIGNED(100000,32));

   TERM4 <= TO_INTEGER(TO_SIGNED((TO_INTEGER(TO_SIGNED((C4*DX*22),32) /
TO_SIGNED(10000,32)) * DY),32) / TO_SIGNED(10000,32));

   TERM5 <= TO_INTEGER(TO_SIGNED((TO_INTEGER(TO_SIGNED((C5*DX*22),32) /
TO_SIGNED(10000,32)) * DZ),32) / TO_SIGNED(10000,32));

   TERM6 <= TO_INTEGER(TO_SIGNED((TO_INTEGER(TO_SIGNED((C6*DY*22),32) /
TO_SIGNED(10000,32)) * DZ),32) / TO_SIGNED(10000,32));

   TERM7 <=
TO_INTEGER(TO_SIGNED((TO_INTEGER(TO_SIGNED((TO_INTEGER(TO_SIGNED((C7*DX)
,32) / TO_SIGNED(10000,32)) * DY),32) / TO_SIGNED(100,32)) * DZ),32) /
TO_SIGNED(10000,32));

   RESULT <= C0 + TERM1 + TERM2 + TERM3 + TERM4 + TERM5 + TERM6 + TERM7;

end BEHAVIORAL;
```

9.2.5 tri_addr_block.vhd

```
-- Description:  Trilinear Address Calculator
-- Filename: tri_addr_block.vhd

library IEEE,DWARE;
use IEEE.numeric_std.all;
use DWARE.DW_Foundation.all;
--library IEEE;
--use IEEE.STD_LOGIC_1164.all;

entity TRI_ADDR_BLOCK is

   port (IN1, IN2, IN3, IN4, IN5, IN6, IN7: in INTEGER;
       MEMADDR: out INTEGER);

end TRI_ADDR_BLOCK;

architecture BEHAVIORAL of TRI_ADDR_BLOCK is
begin

   MEMADDR <= (IN1 + (IN2*IN3)) + ((IN4*IN5) + (IN6*IN7));

end BEHAVIORAL;
```

9.2.6 tri_tb_top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.ALL;

entity TRI_TB_TOP is
end TRI_TB_TOP;

architecture BEHAVIORAL of TRI_TB_TOP is

   component TABLEMEM
      port (CLK: in STD_LOGIC;
           WR: in STD_LOGIC;            -- Read: WR=0, Write: WR=1
           ADDR: in NATURAL;
             DIN: in INTEGER;
           DOUT: out INTEGER);
   end component;

   component TRI_TOP
      port (CLK, START: in STD_LOGIC;
           X0, X1, X2: in INTEGER;
           DONE: out STD_LOGIC;
           Z0, Z1, Z2: out INTEGER;
           -- External Memory Control Signals
           MEMIN: in INTEGER;
           MEMADDR: out INTEGER);
   end component;

   signal INT_CLK: STD_LOGIC := '0';
   signal INT_START: STD_LOGIC := '0';
   signal INT_DONE: STD_LOGIC;
   signal INT_X0, INT_X1, INT_X2, INT_Z0, INT_Z1, INT_Z2: INTEGER;
   signal INT_MEM_DATA, INT_MEM_ADDR: INTEGER;
   signal INT_WR: STD_LOGIC;   -- Not Used
   signal INT_DIN: INTEGER;    -- Not Used

begin

   INT_CLK <= not INT_CLK after 20 ns;

   TEST: process
     file IN_DATA_FILE: text is in "ici.CMYU.out";
     file OUT_DATA_FILE: text is out "tri_top.out";
     variable L: LINE;
     variable X0TEMP, X1TEMP, X2TEMP: INTEGER;
     variable Z0TEMP, Z1TEMP, Z2TEMP: INTEGER;

   begin
     while not endfile(IN_DATA_FILE) loop
       readline(IN_DATA_FILE, L);
       read(L, X0TEMP);
       read(L, X1TEMP);
       read(L, X2TEMP);
```

```
        INT_X0 <= X0TEMP;
        INT_X1 <= X1TEMP;
        INT_X2 <= X2TEMP;
        INT_START <= '1';
        wait until INT_DONE='1';
        Z0TEMP:= INT_Z0;
        Z1TEMP:= INT_Z1;
        Z2TEMP:= INT_Z2;
        write(L, X0TEMP);
        write(L, ' ');
        write(L, X1TEMP);
        write(L, ' ');
        write(L, X2TEMP);
        write(L, ' ');
        write(L, Z0TEMP);
        write(L, ' ');
        write(L, Z1TEMP);
        write(L, ' ');
        write(L, Z2TEMP);
        writeline(OUT_DATA_FILE, L);
      end loop;

      wait;
    end process;

    MEM: TABLEMEM
       port map (CLK=>INT_CLK, WR=>INT_WR, ADDR=>INT_MEM_ADDR,
             DIN=>INT_DIN, DOUT=>INT_MEM_DATA);

    DUT: TRI_TOP
       port map (CLK=>INT_CLK, START=>INT_START,
                X0=>INT_X0, X1=>INT_X1, X2=>INT_X2,
                DONE=>INT_DONE, Z0=>INT_Z0, Z1=>INT_Z1, Z2=>INT_Z2
                MEMIN=>INT_MEM_DATA, MEMADDR=>INT_MEM_ADDR);

end BEHAVIORAL;
```

9.2.7 ici_top.vhd

```
-- Description:  ICI Module Top
-- Filename: ici_top.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ICI_TOP is

   port (CLK, START: in STD_LOGIC;
        -- Input
        X1, X2, X3: in INTEGER;

        -- Parameters
          P1_LO_BOUND, P1_HI_BOUND,
        P2_LO_BOUND, P2_HI_BOUND,
          P3_LO_BOUND, P3_HI_BOUND,
          P_SIZE, K_MAX, EP, MU,
        DEL, INVDEL_OWD, INVDEL_TWD: in INTEGER;

        -- Output
          DONE: out STD_LOGIC;
          INV_OUT0, INV_OUT1, INV_OUT2: out INTEGER;

        -- Memory Module I/O Signals
          MEMIN: in INTEGER;
        MEM_MUX_SEL: out STD_LOGIC;
        MEM_ADDR: out INTEGER;

        -- Trilinear Interpolation Module I/O Signals
        TRI_START: out STD_LOGIC;
          TRI_DONE: in STD_LOGIC;
          TRI_X0, TRI_X1, TRI_X2: out INTEGER;
          TRI_Z0, TRI_Z1, TRI_Z2: in INTEGER);

end ICI_TOP;

architecture STRUCTURAL of ICI_TOP is

   component ICI_CTRL
      port (CLK, START: in STD_LOGIC;
           X1, X2, X3: in INTEGER;
           -- Input Parameters
           P1_LO_BOUND, P1_HI_BOUND,
           P2_LO_BOUND, P2_HI_BOUND,
           P3_LO_BOUND, P3_HI_BOUND,
           P_SIZE, K_MAX, EP,
           DEL, INVDEL_OWD, INVDEL_TWD,
           MU: in INTEGER;
           DONE: out STD_LOGIC;
           INV_OUT0, INV_OUT1, INV_OUT2: out INTEGER;
           -- External Memory Signals
           MEMIN: in INTEGER;
```

```
            MEM_MUX_SEL: out STD_LOGIC;
            -- Memory Address Calculator I/O Signals
            ADDR_CALC_IN1, ADDR_CALC_IN2, ADDR_CALC_IN3: out INTEGER,
              -- Distance Calculator I/O Signals
            D_CALC_X0, D_CALC_X1, D_CALC_X2,
            D_CALC_Y0, D_CALC_Y1, D_CALC_Y2: out INTEGER;
            D_CALC_IN: in INTEGER;
            -- Trilinear Interp. Module I/O Signals
            TRI_START: out STD_LOGIC;
            TRI_DONE: in STD_LOGIC;
            TRI_X0, TRI_X1, TRI_X2: out INTEGER;
            TRI_Z0, TRI_Z1, TRI_Z2: in INTEGER;
              ICI_DP_X0, ICI_DP_X1, ICI_DP_X2,
              ICI_DP_Y0, ICI_DP_Y1, ICI_DP_Y2,
              ICI_DP_GM0, ICI_DP_GM1, ICI_DP_GM2,
              ICI_DP_MU, ICI_DP_INVDEL, ICI_DP_CURRVAL: out INTEGER;
              ICI_DP_NEXT_VAL: in INTEGER);
    end component;

    component ADDR_CALC
       port (IN1, IN2, IN3: in INTEGER;
            ADDR: out INTEGER);
    end component;

    component DIS_DP
       port (X0, X1, X2: in INTEGER;
             Y0, Y1, Y2: in INTEGER;
            DISOUT: out INTEGER);
    end component;

    component ICI_DP
       port (X0, X1, X2, Y0, Y1, Y2, GM0, GM1, GM2, MU, INVDEL,
            CURRVAL: in INTEGER;
            NEXTVAL: out INTEGER);
    end component;

    -- ADDR_CALC internal signals
    signal INT_ADDR_CALC_IN1, INT_ADDR_CALC_IN2, INT_ADDR_CALC_IN3:
INTEGER;

    -- DIS_DP internal signals
    signal INT_D_CALC_X0, INT_D_CALC_X1, INT_D_CALC_X2,
           INT_D_CALC_Y0, INT_D_CALC_Y1, INT_D_CALC_Y2,
           INT_D_CALC_IN: INTEGER;

    -- ICI_DP internal signals
    signal INT_ICI_DP_X0, INT_ICI_DP_X1, INT_ICI_DP_X2,
           INT_ICI_DP_Y0, INT_ICI_DP_Y1, INT_ICI_DP_Y2,
           INT_ICI_DP_GM0, INT_ICI_DP_GM1, INT_ICI_DP_GM2,
           INT_ICI_DP_MU, INT_ICI_DP_INVDEL,
         INT_ICI_DP_CURRVAL, INT_ICI_DP_NEXTVAL: INTEGER;

begin

    CTRL_BLK: ICI_CTRL
       port map (CLK=>CLK,
             START=>START,
```

```
            X1=>X1,
          X2=>X2,
          X3=>X3,
            P1_LO_BOUND=>P1_LO_BOUND,
          P1_HI_BOUND=>P1_HI_BOUND,
            P2_LO_BOUND=>P2_LO_BOUND,
          P2_HI_BOUND=>P2_HI_BOUND,
            P3_LO_BOUND=>P3_LO_BOUND,
          P3_HI_BOUND=>P3_HI_BOUND,
            P_SIZE=>P_SIZE,
          K_MAX=>K_MAX,
          EP=>EP,
            DEL=>DEL,
          INVDEL_OWD=>INVDEL_OWD,
          INVDEL_TWD=>INVDEL_TWD,
            MU=>MU,
            DONE=>DONE,
            INV_OUT0=>INV_OUT0,
          INV_OUT1=>INV_OUT1,
          INV_OUT2=>INV_OUT2,
            MEMIN=>MEMIN,
            MEM_MUX_SEL=>MEM_MUX_SEL,
            ADDR_CALC_IN1=>INT_ADDR_CALC_IN1,
          ADDR_CALC_IN2=>INT_ADDR_CALC_IN2,
          ADDR_CALC_IN3=>INT_ADDR_CALC_IN3,
            D_CALC_X0=>INT_D_CALC_X0,
          D_CALC_X1=>INT_D_CALC_X1,
          D_CALC_X2=>INT_D_CALC_X2,
          D_CALC_Y0=>INT_D_CALC_Y0,
          D_CALC_Y1=>INT_D_CALC_Y1,
          D_CALC_Y2=>INT_D_CALC_Y2,
            D_CALC_IN=>INT_D_CALC_IN,
            TRI_START=>TRI_START,
            TRI_DONE=>TRI_DONE,
            TRI_X0=>TRI_X0,
          TRI_X1=>TRI_X1,
          TRI_X2=>TRI_X2,
            TRI_Z0=>TRI_Z0,
          TRI_Z1=>TRI_Z1,
          TRI_Z2=>TRI_Z2,
              ICI_DP_X0=>INT_ICI_DP_X0,
          ICI_DP_X1=>INT_ICI_DP_X1,
          ICI_DP_X2=>INT_ICI_DP_X2,
          ICI_DP_Y0=>INT_ICI_DP_Y0,
          ICI_DP_Y1=>INT_ICI_DP_Y1,
          ICI_DP_Y2=>INT_ICI_DP_Y2,
              ICI_DP_GM0=>INT_ICI_DP_GM0,
          ICI_DP_GM1=>INT_ICI_DP_GM1,
          ICI_DP_GM2=>INT_ICI_DP_GM2,
              ICI_DP_MU=>INT_ICI_DP_MU,
          ICI_DP_INVDEL=>INT_ICI_DP_INVDEL,
          ICI_DP_CURRVAL=>INT_ICI_DP_CURRVAL,
              ICI_DP_NEXT_VAL=>INT_ICI_DP_NEXTVAL);

ADDR_CALC_BLK: ADDR_CALC
   port map (IN1=>INT_ADDR_CALC_IN1,
             IN2=>INT_ADDR_CALC_IN2,
```

```
            IN3=>INT_ADDR_CALC_IN3,
              ADDR=>MEM_ADDR);

   DIS_DP_BLK: DIS_DP
      port map (X0=>INT_D_CALC_X0,
                X1=>INT_D_CALC_X1,
             X2=>INT_D_CALC_X2,
                Y0=>INT_D_CALC_Y0,
             Y1=>INT_D_CALC_Y1,
             Y2=>INT_D_CALC_Y2,
               DISOUT=>INT_D_CALC_IN);

   ICI_DP_BLK: ICI_DP
      port map (X0=>INT_ICI_DP_X0,
                X1=>INT_ICI_DP_X1,
             X2=>INT_ICI_DP_X2,
             Y0=>INT_ICI_DP_Y0,
             Y1=>INT_ICI_DP_Y1,
             Y2=>INT_ICI_DP_Y2,
             GM0=>INT_ICI_DP_GM0,
             GM1=>INT_ICI_DP_GM1,
             GM2=>INT_ICI_DP_GM2,
             MU=>INT_ICI_DP_MU,
             INVDEL=>INT_ICI_DP_INVDEL,
               CURRVAL=>INT_ICI_DP_CURRVAL,
               NEXTVAL=>INT_ICI_DP_NEXTVAL);

end STRUCTURAL;
```

9.2.8 ici_ctrl.vhd

```
-- Description:  ICI Control Block
-- Filename: ici_ctrl.vhd
-- Revisions: 4/18/02  Creation.

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ICI_CTRL is
   port (CLK, START: in STD_LOGIC;
        X1, X2, X3: in INTEGER;
        -- Input Parameters
        P1_LO_BOUND, P1_HI_BOUND,
        P2_LO_BOUND, P2_HI_BOUND,
        P3_LO_BOUND, P3_HI_BOUND,
        P_SIZE,
        K_MAX,
        EP,
        DEL, INVDEL_OWD, INVDEL_TWD,
        MU: in INTEGER;
        DONE: out STD_LOGIC;
        INV_OUT0, INV_OUT1, INV_OUT2: out INTEGER;

        -- External Memory Signals
        MEMIN: in INTEGER;
        MEM_MUX_SEL: out STD_LOGIC;

        -- Memory Address Calculator I/O Signals
        ADDR_CALC_IN1, ADDR_CALC_IN2, ADDR_CALC_IN3: out INTEGER,

          -- Distance Calculator I/O Signals
        D_CALC_X0, D_CALC_X1, D_CALC_X2,
        D_CALC_Y0, D_CALC_Y1, D_CALC_Y2: out INTEGER;
        D_CALC_IN: in INTEGER;

        -- Trilinear Interp. Module I/O Signals
        TRI_START: out STD_LOGIC;
        TRI_DONE: in STD_LOGIC;
        TRI_X0, TRI_X1, TRI_X2: out INTEGER;
        TRI_Z0, TRI_Z1, TRI_Z2: in INTEGER;

          ICI_DP_X0, ICI_DP_X1, ICI_DP_X2,
          ICI_DP_Y0, ICI_DP_Y1, ICI_DP_Y2,
          ICI_DP_GM0, ICI_DP_GM1, ICI_DP_GM2,
          ICI_DP_MU, ICI_DP_INVDEL, ICI_DP_CURRVAL: out INTEGER;
          ICI_DP_NEXT_VAL: in INTEGER);

end ICI_CTRL;

architecture BEHAVIORAL of ICI_CTRL is

   signal INT_ADD1_RESULT, INT_ADD1A, INT_ADD1B: INTEGER;
```

```
begin

   process
      variable I, J, K: INTEGER;
      type D_ARRAY is array (NATURAL range 0 to 3) of INTEGER;
      type I_ARRAY is array (NATURAL range 0 to 3) of NATURAL;
      variable CLOSED: D_ARRAY;
      variable CLOSEIDX: I_ARRAY;

      variable TEMP_C, TEMP_M, TEMP_Y,
             TEMP_C_I, TEMP_M_I, TEMP_Y_I: INTEGER;

      variable CLOSED_FINAL: INTEGER;

      variable ADD1_IN, ADD1_CONST: INTEGER;
      variable FD_FLAG, BD_FLAG: BOOLEAN;
      variable SUB1_INA, SUB1_INB,
             SUB2_INA, SUB2_INB,
             SUB3_INA, SUB3_INB: INTEGER;
      variable TEMP_GM0, TEMP_GM1, TEMP_GM2,
               TEMP_GM3, TEMP_GM4, TEMP_GM5,
               TEMP_GM6, TEMP_GM7, TEMP_GM8: INTEGER;
      variable CURR_C, CURR_M, CURR_Y: INTEGER;
      variable NEXT_C, NEXT_M, NEXT_Y: INTEGER;
      variable C_INVDEL, M_INVDEL, Y_INVDEL: INTEGER;
      variable HI_BOUND, LO_BOUND: INTEGER;

   begin
      loop
         wait until CLK'event and CLK='1';
         DONE <= '0';
         if START = '0' then
            CLOSED(0) := 2000000000;
            CLOSED(1) := 2000000000;
            CLOSED(2) := 2000000000;
            CLOSED(3) := 2000000000;
            D_CALC_X0 <= X1;
            D_CALC_X1 <= X2;
            D_CALC_X2 <= X3;
         else
            CLOSED(0) := 2000000000;
            CLOSED(1) := 2000000000;
            CLOSED(2) := 2000000000;
            CLOSED(3) := 2000000000;
            D_CALC_X0 <= X1;
            D_CALC_X1 <= X2;
            D_CALC_X2 <= X3;

            -- COMPUTE INITIAL ESTIMATE
            -- FIRST WE MUST GET THE CLOSEST POINTS TO X
            I:=0;
            MEM_MUX_SEL <= '0';

            while I < P_SIZE loop
               ADDR_CALC_IN1 <= 6;
               ADDR_CALC_IN2 <= I;
               ADDR_CALC_IN3 <= 3;
```

```
   wait until CLK'event and CLK='1';

   ADDR_CALC_IN1 <= 6;
   ADDR_CALC_IN2 <= I;
   ADDR_CALC_IN3 <= 4;
   wait until CLK'event and CLK='1';

   D_CALC_Y0 <= MEMIN;
   ADDR_CALC_IN1 <= 6;
   ADDR_CALC_IN2 <= I;
   ADDR_CALC_IN3 <= 5;
   wait until CLK'event and CLK='1';

   D_CALC_Y1 <= MEMIN;
   wait until CLK'event and CLK='1';

   D_CALC_Y2 <= MEMIN;
   wait until CLK'event and CLK='1';
   wait until CLK'event and CLK='1';
   wait until CLK'event and CLK='1';
   wait until CLK'event and CLK='1';
   wait until CLK'event and CLK='1';

   if D_CALC_IN < CLOSED(0) then
      CLOSED(3) := CLOSED(2);
          CLOSED(2) := CLOSED(1);
    CLOSED(1) := CLOSED(0);
    CLOSED(0) := D_CALC_IN;
    CLOSEIDX(3) := CLOSEIDX(2);
          CLOSEIDX(2) := CLOSEIDX(1);
    CLOSEIDX(1) := CLOSEIDX(0);
    CLOSEIDX(0) := I;
   elsif D_CALC_IN < CLOSED(1) then
    CLOSED(3) := CLOSED(2);
    CLOSED(2) := CLOSED(1);
    CLOSED(1) := D_CALC_IN;
    CLOSEIDX(3) := CLOSEIDX(2);
    CLOSEIDX(2) := CLOSEIDX(1);
    CLOSEIDX(1) := I;
   elsif D_CALC_IN < CLOSED(2) then
    CLOSED(3) := CLOSED(2);
    CLOSED(2) := D_CALC_IN;
    CLOSEIDX(3) := CLOSEIDX(2);
    CLOSEIDX(2) := I;
   elsif D_CALC_IN < CLOSED(3) then
    CLOSED(3) := D_CALC_IN;
    CLOSEIDX(3) := I;
   end if;

 I:=I+1;
end loop;

I:=0;
CLOSED_FINAL := 2000000000;
while I < 4 loop
   ADDR_CALC_IN1 <= 6;
   ADDR_CALC_IN2 <= CLOSEIDX(I);
```

```
ADDR_CALC_IN3 <= 0;
wait until CLK'event and CLK='1';

ADDR_CALC_IN1 <= 6;
ADDR_CALC_IN2 <= CLOSEIDX(I);
ADDR_CALC_IN3 <= 1;
wait until CLK'event and CLK='1';

TEMP_C := MEMIN;
ADDR_CALC_IN1 <= 6;
ADDR_CALC_IN2 <= CLOSEIDX(I);
ADDR_CALC_IN3 <= 2;
wait until CLK'event and CLK='1';

TEMP_M := MEMIN;
wait until CLK'event and CLK='1';

TEMP_Y := MEMIN;

K:=0;
while K < 3 loop
   J:=-4;
   while J < 5 loop
      INT_ADD1B <= J*DEL;
      if K=0 then
         INT_ADD1A <= TEMP_C;
      elsif K=1 then
         INT_ADD1A <= TEMP_M;
      else
         INT_ADD1A <= TEMP_Y;
      end if;

      wait until CLK'event and CLK='1';

      if K=0 then
         if INT_ADD1_RESULT > P1_HI_BOUND then
          TEMP_C_I := P1_HI_BOUND;
         elsif INT_ADD1_RESULT < P1_LO_BOUND then
          TEMP_C_I := P1_LO_BOUND;
         else
            TEMP_C_I := INT_ADD1_RESULT;
         end if;
         TEMP_M_I := TEMP_M;
         TEMP_Y_I := TEMP_Y;
      elsif K=1 then
         TEMP_C_I := TEMP_C;
         if INT_ADD1_RESULT > P2_HI_BOUND then
          TEMP_M_I := P2_HI_BOUND;
         elsif INT_ADD1_RESULT < P2_LO_BOUND then
          TEMP_M_I := P2_LO_BOUND;
         else
            TEMP_M_I := INT_ADD1_RESULT;
         end if;
         TEMP_Y_I := TEMP_Y;
      else
         TEMP_C_I := TEMP_C;
         TEMP_M_I := TEMP_M;
```

```
            if INT_ADD1_RESULT > P3_HI_BOUND then
             TEMP_Y_I := P3_HI_BOUND;
            elsif INT_ADD1_RESULT < P3_LO_BOUND then
             TEMP_Y_I := P3_LO_BOUND;
            else
               TEMP_Y_I := INT_ADD1_RESULT;
            end if;
         end if;

         TRI_X0 <= TEMP_C_I;
         TRI_X1 <= TEMP_M_I;
         TRI_X2 <= TEMP_Y_I;
         TRI_START <= '1';
         MEM_MUX_SEL <= '1';

         loop
            wait until CLK'event and CLK='1';
            TRI_START <= '0';
            if (TRI_DONE = '1') then
             MEM_MUX_SEL <= '0';
             exit;
            end if;
         end loop;

         D_CALC_X0 <= X1;
         D_CALC_X1 <= X2;
         D_CALC_X2 <= X3;
         D_CALC_Y0 <= TRI_Z0;
         D_CALC_Y1 <= TRI_Z1;
         D_CALC_Y2 <= TRI_Z2;
         wait until CLK'event and CLK='1';
         wait until CLK'event and CLK='1';
         wait until CLK'event and CLK='1';
         wait until CLK'event and CLK='1';
         wait until CLK'event and CLK='1';

         if D_CALC_IN < CLOSED_FINAL then
            CURR_C := TEMP_C_I;
            CURR_M := TEMP_M_I;
            CURR_Y := TEMP_Y_I;
            CLOSED_FINAL := D_CALC_IN;
         end if;

         J:=J+1;
      end loop;
      K:=K+1;
  wait until CLK'event and CLK='1';
   end loop;
I:=I+1;
end loop;

-- INITIAL ESTIMATE AVAILABLE

-- COMPUTE GRADIENT MATRIX
I:=0;
while (I<=2) loop -- Gradient Matrix
   if (I=0) then
```

```
    ADD1_IN := CURR_C;
  HI_BOUND := P1_HI_BOUND;
  LO_BOUND := P1_LO_BOUND;
   elsif (I=1) then
    ADD1_IN := CURR_M;
  HI_BOUND := P2_HI_BOUND;
  LO_BOUND := P2_LO_BOUND;
   else
    ADD1_IN := CURR_Y;
  HI_BOUND := P3_HI_BOUND;
  LO_BOUND := P3_LO_BOUND;
   end if;

INT_ADD1A <= HI_BOUND;
INT_ADD1B <= -DEL;
  wait until CLK'event and CLK='1';

  if (ADD1_IN <= INT_ADD1_RESULT) then
   INT_ADD1B <= DEL;
   FD_FLAG := TRUE;
  else
   INT_ADD1B <= 0;
   FD_FLAG := FALSE;
  end if;

INT_ADD1A <= ADD1_IN;
  wait until CLK'event and CLK='1';

  if (I=0) then
     TRI_X0 <= INT_ADD1_RESULT;
     TRI_X1 <= CURR_M;
     TRI_X2 <= CURR_Y;
  elsif (I=1) then
     TRI_X0 <= CURR_C;
     TRI_X1 <= INT_ADD1_RESULT;
     TRI_X2 <= CURR_Y;
  else
     TRI_X0 <= CURR_C;
     TRI_X1 <= CURR_M;
     TRI_X2 <= INT_ADD1_RESULT;
  end if;

INT_ADD1A <= LO_BOUND;
INT_ADD1B <= DEL;

  TRI_START <= '1';
  MEM_MUX_SEL <= '1';
  loop
     wait until CLK'event and CLK='1';
     TRI_START <= '0';
     if (TRI_DONE = '1') then
      MEM_MUX_SEL <= '0';
      exit;
     end if;
  end loop;

  SUB1_INA := TRI_Z0;
```

```
  SUB2_INA := TRI_Z1;
  SUB3_INA := TRI_Z2;

  if (ADD1_IN >= INT_ADD1_RESULT) then
   INT_ADD1B <= -DEL;
   BD_FLAG := TRUE;
  else
   INT_ADD1B <= 0;
   BD_FLAG := FALSE;
  end if;

INT_ADD1A <= ADD1_IN;
  wait until CLK'event and CLK='1';

  if (I=0) then
     TRI_X0 <= INT_ADD1_RESULT;
     TRI_X1 <= CURR_M;
     TRI_X2 <= CURR_Y;
  elsif (I=1) then
     TRI_X0 <= CURR_C;
     TRI_X1 <= INT_ADD1_RESULT;
     TRI_X2 <= CURR_Y;
  else
     TRI_X0 <= CURR_C;
     TRI_X1 <= CURR_M;
     TRI_X2 <= INT_ADD1_RESULT;
  end if;

  TRI_START <= '1';
  MEM_MUX_SEL <= '1';
  loop
     wait until CLK'event and CLK='1';
     TRI_START <= '0';
     if (TRI_DONE = '1') then
      MEM_MUX_SEL <= '0';
      exit;
     end if;
  end loop;

SUB1_INB := TRI_Z0;
  SUB2_INB := TRI_Z1;
  SUB3_INB := TRI_Z2;

  if I=0 then
     TEMP_GM0 := SUB1_INA - SUB1_INB;
     TEMP_GM1 := SUB2_INA - SUB2_INB;
     TEMP_GM2 := SUB3_INA - SUB3_INB;
     if FD_FLAG = TRUE and BD_FLAG = TRUE then
        C_INVDEL := INVDEL_TWD;
     else
        C_INVDEL := INVDEL_OWD;
     end if;
  elsif I=1 then
     TEMP_GM3 := SUB1_INA - SUB1_INB;
     TEMP_GM4 := SUB2_INA - SUB2_INB;
     TEMP_GM5 := SUB3_INA - SUB3_INB;
     if FD_FLAG = TRUE and BD_FLAG = TRUE then
```

```
            M_INVDEL := INVDEL_TWD;
         else
            M_INVDEL := INVDEL_OWD;
         end if;
   else
         TEMP_GM6 := SUB1_INA - SUB1_INB;
         TEMP_GM7 := SUB2_INA - SUB2_INB;
         TEMP_GM8 := SUB3_INA - SUB3_INB;
         if FD_FLAG = TRUE and BD_FLAG = TRUE then
            Y_INVDEL := INVDEL_TWD;
         else
            Y_INVDEL := INVDEL_OWD;
         end if;
     end if;

   I:=I+1;
  end loop;
-- GRADIENT MATRIX AVAILABLE

  -- BEGIN ICI ITERATION
J:=0;
while J <= K_MAX loop
     TRI_X0 <= CURR_C;
     TRI_X1 <= CURR_M;
     TRI_X2 <= CURR_Y;
     TRI_START <= '1';
     MEM_MUX_SEL <= '1';

     loop
        wait until CLK'event and CLK='1';
        TRI_START <= '0';
        if (TRI_DONE = '1') then
         MEM_MUX_SEL <= '0';
         exit;
        end if;
     end loop;

     D_CALC_X0 <= X1;
     D_CALC_X1 <= X2;
     D_CALC_X2 <= X3;
     D_CALC_Y0 <= TRI_Z0;
     D_CALC_Y1 <= TRI_Z1;
     D_CALC_Y2 <= TRI_Z2;
     wait until CLK'event and CLK='1';
     wait until CLK'event and CLK='1';
     wait until CLK'event and CLK='1';
     wait until CLK'event and CLK='1';
     wait until CLK'event and CLK='1';
     if D_CALC_IN <= EP then
        INV_OUT0 <= CURR_C;
        INV_OUT1 <= CURR_M;
        INV_OUT2 <= CURR_Y;
        exit;
     else

      I:=0;
        while I < 3 loop
```

```
    ICI_DP_X0 <= X1;
    ICI_DP_X1 <= X2;
    ICI_DP_X2 <= X3;
    ICI_DP_Y0 <= TRI_Z0;
    ICI_DP_Y1 <= TRI_Z1;
    ICI_DP_Y2 <= TRI_Z2;
    ICI_DP_MU <= MU;

if I=0 then
       ICI_DP_GM0 <= TEMP_GM0;
       ICI_DP_GM1 <= TEMP_GM1;
       ICI_DP_GM2 <= TEMP_GM2;
       ICI_DP_INVDEL <= C_INVDEL;
       ICI_DP_CURRVAL <= CURR_C;
    elsif I=1 then
   ICI_DP_GM0 <= TEMP_GM3;
   ICI_DP_GM1 <= TEMP_GM4;
   ICI_DP_GM2 <= TEMP_GM5;
       ICI_DP_INVDEL <= M_INVDEL;
       ICI_DP_CURRVAL <= CURR_M;
    else
   ICI_DP_GM0 <= TEMP_GM6;
   ICI_DP_GM1 <= TEMP_GM7;
   ICI_DP_GM2 <= TEMP_GM8;
       ICI_DP_INVDEL <= Y_INVDEL;
       ICI_DP_CURRVAL <= CURR_Y;
end if;

    wait until CLK'event and CLK='1';
    wait until CLK'event and CLK='1';
    wait until CLK'event and CLK='1';
    wait until CLK'event and CLK='1';

    if I=0 then
       if ICI_DP_NEXT_VAL > P1_HI_BOUND then
        CURR_C := P1_HI_BOUND;
       elsif ICI_DP_NEXT_VAL < P1_LO_BOUND then
        CURR_C := P1_LO_BOUND;
       else
          CURR_C := ICI_DP_NEXT_VAL;
       end if;
    elsif I=1 then
       if ICI_DP_NEXT_VAL > P2_HI_BOUND then
        CURR_M := P2_HI_BOUND;
       elsif ICI_DP_NEXT_VAL < P2_LO_BOUND then
        CURR_M := P2_LO_BOUND;
       else
          CURR_M := ICI_DP_NEXT_VAL;
       end if;
    else
       if ICI_DP_NEXT_VAL > P3_HI_BOUND then
        CURR_Y := P3_HI_BOUND;
       elsif ICI_DP_NEXT_VAL < P3_LO_BOUND then
        CURR_Y := P3_LO_BOUND;
       else
          CURR_Y := ICI_DP_NEXT_VAL;
       end if;
```

```
                    end if;

                  I:=I+1;
                   end loop;
                end if;

              J:=J+1;
             end loop;

           INV_OUT0 <= CURR_C;
           INV_OUT1 <= CURR_M;
           INV_OUT2 <= CURR_Y;
             DONE <= '1';

         end if; -- start
      end loop; -- process loop

   end process;

   -- Shared utility adder used by ctrl block
   INT_ADD1_RESULT <= INT_ADD1A + INT_ADD1B;

end BEHAVIORAL;
```

9.2.9 ici_dp.vhd

```
-- Description:  Main ICI Loop Datapath
-- Filename: ici_dp.vhd
-- Revisions: 4/13/02  Creation.

library IEEE, DWARE;
--library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use DWARE.DW_Foundation.all;

entity ICI_DP is

   port (X0, X1, X2, Y0, Y1, Y2, GM0, GM1, GM2, MU, INVDEL, CURRVAL: in
INTEGER;
       NEXTVAL: out INTEGER);

end ICI_DP;

architecture BEHAVIORAL of ICI_DP is

   signal TEMP_0, TEMP_1, TEMP_2, TEMP_M1: INTEGER;

begin

   TEMP_0 <= (X0 - Y0) * GM0;
   TEMP_1 <= (X1 - Y1) * GM1;
   TEMP_2 <= (X2 - Y2) * GM2;

   TEMP_M1 <= MU * INVDEL;

   NEXTVAL <= CURRVAL + TO_INTEGER(TO_SIGNED(((TEMP_0 + TEMP_1 + TEMP_2)
* TEMP_M1), 32) / TO_SIGNED(1000000, 32));

end BEHAVIORAL;
```

9.2.10 dis_dp.vhd

```
-- Description:  Distance Calculator Datapath
-- Filename: dis_dp.vhd
-- Revisions: 4/13/02  Creation.

library IEEE, DWARE;
--library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use IEEE.MATH_REAL.all;
use DWARE.DW_Foundation.all;

entity DIS_DP is

   port (X0, X1, X2: in INTEGER;
         Y0, Y1, Y2: in INTEGER;
        DISOUT: out INTEGER);

end DIS_DP;

architecture BEHAVIORAL of DIS_DP is

   signal INT_D1, INT_D2, INT_D3: INTEGER;

begin

   INT_D1 <= X0 - Y0;
   INT_D2 <= X1 - Y1;
   INT_D3 <= X2 - Y2;

   DISOUT <= TO_INTEGER(SQRT(TO_SIGNED(((INT_D1*INT_D1) +
(INT_D2*INT_D2) + (INT_D3*INT_D3)),32)));

   -- For simulation purposes
--   DISOUT <= INTEGER(SQRT(REAL((INT_D1*INT_D1) + (INT_D2*INT_D2) +
(INT_D3*INT_D3))));

end BEHAVIORAL;
```

9.2.11 addr_calc.vhd

```
-- Description:  Memory Address Calculator Datapath
-- Filename: addr_calc.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ADDR_CALC is

   port (IN1, IN2, IN3: in INTEGER;
       ADDR: out INTEGER);

end ADDR_CALC;

architecture BEHAVIORAL of ADDR_CALC is
begin

   ADDR <= (IN1 * IN2) + IN3;

end BEHAVIORAL;
```

9.2.12 tablemem.vhd

```
-- Description: Memory module to hold the look-up table.
-- Filename: tablemem.vhd
-- Revisions: 3/13/02  Creation.
-- NOTE: This memory module is not intended for synthesis.

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use STD.TEXTIO.ALL;

entity TABLEMEM is
  generic (MEMSIZE: NATURAL := (6*2197));

  port (CLK: in STD_LOGIC;
      WR: in STD_LOGIC;           -- Read: WR=0, Write: WR=1
      ADDR: in NATURAL;
        DIN: in INTEGER;
      DOUT: out INTEGER);

end TABLEMEM;

architecture BEHAVIORAL of TABLEMEM is
  type MEM_ARRAY is array (NATURAL range 0 to MEMSIZE) of INTEGER;

begin

   LOAD: process
     file LUT1: text is in "cmy.dat";
     file LUT2: text is in "lab.dat";
     variable L: LINE;
     variable IDX: INTEGER;
     variable MEM: MEM_ARRAY;

   begin
     IDX := 0;
     while not endfile(LUT1) loop
       -- Read from CMY
       readline(LUT1, L);
       read(L, MEM(IDX));
       IDX := IDX + 1;
       read(L, MEM(IDX));
       IDX := IDX + 1;
       read(L, MEM(IDX));
       IDX := IDX + 1;

       -- Read from LAB
       readline(LUT2, L);
       read(L, MEM(IDX));
       IDX := IDX + 1;
       read(L, MEM(IDX));
       IDX := IDX + 1;
       read(L, MEM(IDX));
```

```
        IDX := IDX + 1;
      end loop;

      loop
        if (WR = '1') then
           MEM(ADDR) := DIN;
        else
         if (ADDR >= 0) then
            DOUT <= MEM(ADDR);
           end if;
        end if;
        wait until CLK'event and CLK='1';
      end loop;
    end process LOAD;

end BEHAVIORAL;
```